National Soil Information System (NASIS)

CVIR Script Writing

NASIS Calculation/Validation, Interpretation and Reporting (CVIR) functions give you the capability to select data from the NASIS database, manipulate and format the data in virtually any way you want, print the data on a page, save it in a file, or populate the database. NASIS uses a *CVIR scripting language* to give you this capability. Scripts range from simple to complex. They are used to produce calculations, validations, properties, and all kinds of reports from the "Where Used" report you see in the lower pane of your NASIS window to manuscript tables and "MUG" reports.

The NASIS CVIR scripting language includes a common set of commands used by reports, calculations, validations, and properties. The common set of commands used in NASIS makes it easier for you to learn new functions by building on what you already know.

Understanding and using the scripting language is the main focus of the NASIS CVIR script writing workshops,but the emphasis is on writing reports. At the conclusion of the Basic Report Writing Workshop, you should be able to create simple to moderately complex report scripts for everyday use and understand the principles used in creating reports. At the conclusion of the Advanced Report Writing Workshop, you should be able to create moderately complex reports and understand data collection and array manipulation, data structures used in interpretations, and techniques and principles used to calculations, validations and properties. With practice after these workshops, you should be able to create scripts to meet a wide variety of needs.

Contents

Understanding NASIS Reports

A report is produced by running a previously written report script. The script contains a number of statements which define a database "view" and specify how the data in the view will be placed on the report page. A database view is a set of database columns derived from one or more queries of the selected set or permanent database. In addition, data can be brought into a report from properties, interpretations, or data files, and new data values may be calculated in the report script. Once all these data are assembled the report moves into the output processing phase, which is controlled by report "templates" and "sections". A template defines how a line of the report is laid out, and a section defines when that line appears in the report. By carefully defining the templates and sections, very fine control of the report format can be achieved.

Report Scripts

Report scripts are stored in NASIS as part of the Report object, and are created with the text editor. Scripts can be edited only by a person who is a member of the group that owns the report.

A report script consists of statements from the report scripting language defined in this manual. The simplest type of report script contains only a single EXEC SQL statement. The default output format prints each data element in the SELECT clause as a report column. The column heading, width, and format come from the data dictionary, so they appear as they do in the NASIS editor. You can use a complex query, including joins between multiple tables, but the default format is of limited use because no properties or defined variables can be printed. Also, text fields do not print well because their default width is too narrow. More complete report scripts manipulate the database view, derive additional data, and specify page layouts.

Report Script Organization

Report scripts are "non-procedural", which means that they do not describe a step-by-step procedure for producing a report. The procedure is built into the report generator, and the script defines the rules that control how each piece of data is derived and formatted. This means that there is flexibility in the order that statements appear in a script, subject to the following restrictions and conventions.

- A statement that defines a name for a variable *must* precede a statement that uses the variable in a formula or parameter list. Likewise, section and template names must be defined before they can be used in section or report line specifications.
- When the report script executes, all queries are executed first, then all derivations, and finally all variable expressions are computed. Within each of these three groups, the statements are performed in the order written.
- Report sections are evaluated for possible output in the order written.
- Each statement in the report language ends with a period.
- The # symbol normally begins a comment, which extends to the end of the line, except in the case of the PARAMETER statement.
- For best readability of the script the suggested order for use of statements is PARAMETER, BASE TABLE, ACCEPT, INTERPRET, EXEC SQL, DERIVE, DEFINE & ASSIGN, PAGE, MARGIN, PITCH, FONT, HEADER & FOOTER, TEMPLATE, and SECTION.

NASIS CVIR Script Statements

Conventions used in this Guide

Report script statements are arranged alphabetically in this technical guide so you can find them more easily. Each statement begins on a new page and continues for as many pages as necessary to describe the statement. The description consists of the following parts:

Syntax: The syntax is described in a formal notation, using the following conventions:

- Braces { } enclose a set of alternatives, of which one must be chosen.
- Any portion of a definition in brackets [ ] is optional.
- When an ellipsis (...) follows the brackets, the optional part can be repeated.
- The symbol ⇒ means "is defined as", and defines a term that appears in a previous statement definition.
- Punctuation in bold print is a required part of the statement.
- Keywords in the report scripting language are shown in sans-serif capital letters (like DEFINE), but the report interpreter is not case sensitive for keywords or variable names.

Used In: The Used In line identifies the type of scripts in which the statement may be used. The types are Report, Property, and Calculation (Validation scripts use the same statements as Calculations).
Example: One or more examples of the statement are shown, followed by an explanation.

Syntax:
ACCEPT variable [, variable] ... **.**

variable ⇒ *name*

Used In:
Report, Property, Calculation

Example:
```
ACCEPT datamapunit_iid.
ACCEPT top_limit, bottom_limit.
```

The ACCEPT statement defines variables that are passed into the script. These variables can be used in expressions to calculate values for other variables. They can also be used in the WHERE clause of a query by writing $name, where name is the name of the variable. This creates a parametric query, as discussed in the next sections.

The first example of the ACCEPT statement could be used in a subreport or a dynamic "Where Used" report. For a subreport, the value of a key column such as datamapunit_iid might be passed by a higher level report, and the subreport would use it in a query to find data related the to data mapunit being processed in the higher level report. In a dynamic report, NASIS automatically copies values from the current row of the base table to the variables in the ACCEPT list, which must be columns in the base table. The ACCEPT list may use either the logical name or the implementation name for a column, but it must be used consistently throughout the report. If a modal column is used, its suffix (such as "_l", "_r", or "_h") must also be used, even with the logical name. The value from each column is converted into either a character string or a floating point number, as described for queries.

The second example might be used in a subreport or a Property script. In this case the variables in the ACCEPT list get their values from the parameters passed by another script that calls the subreport or property. Any variable names may be used because they do not refer to database columns. The number of parameters passed by the caller must equal the number of variables in the ACCEPT list. The type and dimension of these variables are not predefined, so they are determined by the values passed by the caller. For a Property, the primary key columns of the base table act as if they were in the ACCEPT list, even if the property script has no ACCEPT statement. They are used to ensure that the property is providing data for the same record as the script that calls it. Consequently, a calling script and its called property scripts must use the same base table.

Syntax:
BASE TABLE *table-name* **.**

Used In:
Report, Property, Calculation

Example:
```
BASE TABLE component.
```

The BASE TABLE statement specifies which database table controls the operation of the script. For example, a Calculation script operates by performing one iteration for each base table row that has been selected by the user and is in an updateable condition (not locked, protected, etc.). Key input values come from the base table, and the values for calculated data elements are stored in the current row of the base table. Because they only operate on the selected set, CVIR functions always access the base table in the edit tables maintained by the NASIS editor.

A report's iteration is normally based on the first report query, but the base table provides the key used to synchronize queries and properties. A report script requires a base table if the script includes an ACCEPT statement, more than one query, or DERIVE statements.

The *table-name* used in the BASE TABLE statement must be one of the editable tables in the NASIS database. Either the logical name or the implementation name may be used. The base table is also recorded in the data dictionary entry for a Calculation or Validation script, so the two must match. Do not use the "e_" prefix to refer to the base table.

When a base table is declared, the columns containing the table's unique identifier are added internally to the input query, to be used for synchronization between scripts and queries. If a record identifier column is needed in the script, it must be listed by name in the query, because the identifier added for the base table is not accessible by the script.

Syntax:
DEFINE variable [expression] [initialization] **.**
ASSIGN variable expression **.**

expression *(see next page)*
initialization *(see next page)*

Used In:
Report, Property, Calculation

Example:
```
DEFINE status CODENAME (mustatus).
ASSIGN status status || " mapunit".
```

The DEFINE statement defines a variable for use in a CVIR script. Each DEFINE'd variable name must be unique within a script, and must be different from the names of columns in the input query. The ASSIGN statement recalculates the value of a variable that was defined in a previous DEFINE, DERIVE, or EXEC SQL statement. Names may be any combination of letters, numbers and the underscore character, provided that the name starts with a letter and is not the same as one of the reserved words in the language (reserved words are in sans-serif CAPITALS in this document).

Names of variables in different scripts are independent, even if one script calls another via a DERIVE statement. There is one restriction on names in properties that are used in interpretations. Evaluations take their input data from variables named "low", "rv", and "high" (or just "rv" if the property modality is RV). A property called by an evaluation can use other variables for intermediate results, but has to place its final results in variables with these names.

Each variable may have an expression that determines its value. An expression may be based on literals, columns from the input, or other variables. On each iteration of the input, all variables are recalculated in the order that they appear in the DEFINE and ASSIGN statements. Variables are not explicitly typed, so the type is determined by the result of the expression.

An initial value for a variable can also be specified in the DEFINE statement. A variable defined with an initial value and no expression is simply a constant; its value will not be changed.

An important use for an initial value is with an expression that contains the variable being defined. Consider the statement: DEFINE list (list || name) INITIAL "Names: ". This takes the column "name" from each input record and concatenates it to the variable "list", following the initial string "Names: ". If no initial value is defined with this type of expression, the variable starts out with an undefined value, which could produce unpredictable results.

Storing Multiple Values in a Variable

A variable may hold a single value or multiple values, depending on how it is used. The number of values that a variable holds is called its *dimension*. Multiple valued variables are sometimes referred to as *arrays*. Multiple values can be stored for an input column via an AGGREGATE clause. Array variables can also be returned from property calls. Depending on the properties and aggregation types used, variables in the same report can end up with different dimensions. Some operators, such as LOOKUP and WTAVG, can cause a report to fail if the dimensions of their arguments are not the same. So attention must be paid to the way multiple valued variables are processed.

Most of the operators used in expressions do not change the dimension of the data. If an operator uses two or more variables of different dimensions, the result will generally have the highest dimension of the arguments. For example, multiplying a variable with values (1,2,3,4) by the single value 5 produces the multiple valued result (5,10,15,20).

A query that finds no rows results in variables with a dimension of 0, which are typically treated the same way as null values. If all the arguments to an operator have dimension 0 the result will also have dimension 0, but if there is a mixture of zero and non-zero dimensions, the result has the higher dimension. In the above example, multiplying the array (1,2,3,4) by a variable with no values would produce an array of four nulls.

Operators that do not follow these rules are noted in the individual descriptions below. Examples are the array operators like ARRAYSUM that reduce an array of values to a single value.

Expression Syntax

The following syntax rules define all the types of expressions that may be created.

$$\text{expression} \Rightarrow \left\{ \begin{array}{l} \text{literal} \\ \text{element} \\ \text{variable} \\ \text{arithmetic\_expression} \\ \text{conditional\_expression} \\ \text{boolean\_expression} \\ \text{string\_expression} \\ \text{regroup\_expression} \\ \text{function} \\ \textbf{(}\ \text{expression}\ \textbf{)} \end{array} \right\}$$

$\text{initialization} \Rightarrow \text{INITIAL literal}$

$\text{literal} \Rightarrow \{\ \textit{number}\ \mid\ \textbf{“}\textit{string}\textbf{”}\ \}$

$$\text{arithmetic\_expression} \Rightarrow \left\{\begin{array}{l} \text{expression } \{ + \mid \textbf{-} \mid * \mid / \mid ** \} \text{ expression} \\ - \ \text{expression} \end{array}\right\}$$

$$\text{conditional\_expression} \Rightarrow \left\{\begin{array}{l} \text{expression } ? \text{ expression } : \text{ expression} \\ [\ \text{IF}\ ]\ \text{expression THEN expression ELSE expression} \end{array}\right\}$$

$$\text{boolean\_expression} \Rightarrow \left\{\begin{array}{l} \text{comparison} \\ \text{ISNULL } \textbf{(} \text{ expression } \textbf{)} \\ \text{NOT boolean\_expression} \\ \text{ANY expression} \\ \text{ALL expression} \\ \textbf{(} \text{ boolean\_expression } \textbf{)} \\ \text{boolean\_expression } \{ \text{AND} \mid \text{OR} \} \text{ boolean\_expression} \end{array}\right\}$$

$$\text{comparison} \Rightarrow \text{expression} \left\{\begin{array}{c} == \\ != \\ < \\ > \\ <= \\ >= \\ \text{MATCHES} \end{array}\right\} \text{expression}$$

$$\text{string\_expression} \Rightarrow \left\{\begin{array}{l} \text{expression } \textbf{[} \ \textit{number} \ \textbf{:} \ \textit{number} \ \textbf{]} \\ \text{expression } \textbf{||} \text{ expression} \\ \text{CLIP } \textbf{(}\text{expression}\textbf{)} \\ \text{UPCASE } \textbf{(}\text{expression}\textbf{)} \\ \text{LOCASE } \textbf{(}\text{expression}\textbf{)} \\ \text{NMCASE } \textbf{(}\text{expression}\textbf{)} \\ \text{SECASE } \textbf{(}\text{expression}\textbf{)} \\ \text{TEXTURENAME } \textbf{(}\text{expression}\textbf{)} \\ \text{GEOMORDESC } \textbf{(}\text{expression, expression, expression}\textbf{)} \\ \text{STRUCTPARTS } \textbf{(}\text{expression, expression, expression}\textbf{)} \\ \text{ARRAYCAT } \textbf{(}\text{expression, delimiter}\textbf{)} \end{array}\right\}$$

$$\text{regroup\_expression} \Rightarrow \begin{array}{l} \text{REGROUP expression BY expression} \\ \text{AGGREGATE aggregate\_function} \end{array}$$

$$
\text{function} \Rightarrow \left\{ \begin{array}{l}
\text{NEW (expression)} \\
\text{CODENAME (expression [, name])} \\
\text{CODELABEL (expression [, name])} \\
\text{ARRAYSUM (expression)} \\
\text{ARRAYCOUNT (expression)} \\
\text{ARRAYAVG (expression)} \\
\text{ARRAYMIN (expression)} \\
\text{ARRAYMAX (expression)} \\
\text{ARRAYMEDIAN (expression)} \\
\text{ARRAYMODE (expression)} \\
\text{ARRAYSTDEV(expression)} \\
\text{ARRAYSHIFT (expression, expression)} \\
\text{ARRAYROT (expression, expression)} \\
\text{LOOKUP (expression, expression, expression)} \\
\text{WTAVG (expression, expression)} \\
\text{SPRINTF ("}\mathit{string}\text{", expression [, expression] ...)} \\
\text{TODAY} \\
\text{USER} \\
\text{`}\mathit{unix\_command}\text{`} \\
\text{SUM (expression)} \\
\text{COUNT (expression)} \\
\text{AVERAGE (expression)} \\
\text{MIN (expression)} \\
\text{MAX (expression)} \\
\text{LOGN (expression)} \\
\text{LOG10 (expression)} \\
\text{EXP (expression)} \\
\text{COS (expression)} \\
\text{SIN (expression)} \\
\text{TAN (expression)} \\
\text{ACOS (expression)} \\
\text{ASIN (expression)} \\
\text{ATAN (expression)} \\
\text{ATAN2 (expression, expression)} \\
\text{SQRT (expression)} \\
\text{ABS (expression)} \\
\text{POW (expression, expression)} \\
\text{MOD (expression, expression)} \\
\text{ROUND (expression [, expression])}
\end{array} \right\}
$$

Explanation of Expression Syntax

An expression can produce either a numeric or a character string value, depending on its contents. Numeric and character data can be mixed in expressions, with type conversions similar to those done by Informix. Evaluation of expressions follows C conventions for operator precedence. For example, the arithmetic expression: `A + B * C` is evaluated as `A + (B * C)` because multiplication has higher precedence than addition.

Most of the expressions involving arithmetic, boolean and comparison operators require little explanation. They work as would be expected, and produce numeric results. The operator ** denotes exponentiation; the expression `A ** B` is equivalent to the function `POW(A, B)`. Comparisons and boolean expressions produce a 1 for True and a 0 for false.

If a null value is used in an expression, the result is normally null. However, in comparisons, a null value is treated as less than any non-null value and two nulls are considered equal to each other. In boolean expressions, a null is considered False. Invalid computations, such as division by zero, produce a null result. Special cases with null values are noted individually.

String Expressions

String expressions allow for substring extraction, string concatenation, and case changes. They expect to operate on character type input, and will convert the input to character if necessary. Note that when a number is converted to a string it is expressed with 6 decimal places. To produce different formats for numbers, use the SPRINTF function. The results of the following string expressions are always character strings:

***expression* [*n1*:*n2*]**
Returns a substring of the string expression, starting at position *n1* for a length of *n2* characters. The first character of the string is position 0. Note, this differs from the way substrings are defined in Informix queries.
Example: if variable A has the value “Sample”, the expression A[1:3] returns the value “amp”.

expression* || *expression
Concatenates two strings.
Example: the expression “ABC” || “DEF” produces the string “ABCDEF”. If one expression in a concatenation is null it is treated as the string “”, so the result is not a null value unless both of the expressions are null.

CLIP (*expression*)
Removes trailing blanks from a string. This is not normally necessary because NASIS removes trailing blanks when reading data from the database.
Example: the expression CLIP(“ABC   ”) produces the string “ABC”.

UPCASE (*expression*)
Converts a string to upper case.
Example: the expression UPCASE("ABc12") produces the string "ABC12".

LOCASE (*expression*)
Converts a string to lower case.
Example: the expression LOCASE("ABc12") produces the string "abc12".

NMCASE (*expression*)
Converts a string to "name" case: first letter of each word upper case and the remainder lower case.
Example: the expression NMCASE("now is the time") produces the string "Now Is The Time".

SECASE (*expression*)
Converts a string to "sentence" case: first letter of the string upper case and the remainder lower case.
Example: the expression SECASE("now is the time") produces the string "Now is the time".

TEXTURENAME (*expression*)
Converts a set of texture codes to a special string format used in reports. The expression operated on by TEXTURENAME can have zero or more values, each of which is a string as used in the NASIS data element "texture". This element can contain a mixture of codes for texture classes, modifiers, and terms used in lieu of texture. The codes are expanded and concatenated together, with commas as necessary, to produce a texture description as used in manuscript reports.
Example: if the variable T has two values, one of which is "SL", and the other is "SR- CL GR-SIL", the expression TEXTURENAME(T) produces a result with two values, the string "sandy loam", and the string "stratified clay loam to gravelly silt loam".

GEOMORDESC (*expression*, *expression*, *expression*)
Converts data from the component geomorphic description to a standard landform description string for use in reports. The three expressions used as input can be arrays, but all must have the same number of values. The first parameter is the feature name or names for a component, the second has the feature Id for each feature, and the third has the Exists-On reference for each feature. Where an Exists-On reference matches a feature ID, the two names are combined with the word "on". If two features have the same feature ID the Exists-On reference is attached to both and they are output as separate strings. Other features that do not have an Exists-On relationship are output as separate strings. The number of values in the result can be more or less than the number of values in the input expressions.
Example: Data for this operation would be obtained by joining the component geomorphic description table and the geomorphic feature table, such as:

```
EXEC SQL
SELECT geomorph_feat_name, geomorphic_feat_id, exists_on_feature
FROM component, component_geomorph_desc, real geomorph_feature
WHERE JOIN component TO component_geomorph_desc
AND JOIN component_geomorph_desc TO geomorph_feature;
AGGREGATE COLUMN geomorph_feat_name NONE, geomorphic_feat_id
NONE, exists_on_feature NONE.
```

Assume this query produces the data shown in the following table:

geomorph_feat_name	geomorphic_feat_id	exists_on_feat
alluvial fan		
till plain	1	
pothole	2	1

The expression GEOMORDESC(geomorph_feat_name, geomorphic_feat_id, exists_on_feat) would produce a result with two values, "alluvial fan" and "pothole on till plain".

STRUCTPARTS (*expression, expression, expression*)
Converts data from the Pedon Horizon Soil Structure table to a standard structure description string for use in reports The parameters are used in the same manner as the GEOMORDESC function above. The first parameter would be the type of structure, usually a string concatenated from *structure_grade*, *structure_size*, and *structure_type*. The second parameter is the row identifier, *structure_id*, and the third parameter is the reference column, *structure_parts_to*. The only difference between GEOMORDESC and STRUCTPARTS is that the latter uses the words "parting to" to separate linked structures, instead of "on".

ARRAYCAT (*expression, delimiter*)
Concatenates the values in a multiple valued variable or expression, to produce a single valued result. The first argument is a multiple valued expression, and the second argument is a string to be used as a delimiter between the values. An empty string may be specified as the delimiter. If any values of the first argument are null, they and their associated delimiters are skipped. The result has dimension 0 if the first argument has dimension 0, otherwise it has dimension 1. **Example**: If the variable A has four values, "A1", "A2", Null, and "A4", the expression ARRAYCAT (A, "-") would produce a single string: "A1-A2-A4".

Function Expressions
The following function expressions can use either character or numeric values, and produce results in the same type as the input, unless specified otherwise.

NEW (*expression*)
Returns True if the value of the expression is different from the value it had in the previous iteration of the script, or False if the value is the same as last time.

Example: the expression NEW (mapunit_symbol) would be True each time the mapunit symbol changed.

CODENAME (*expression* [*, name*])
Returns the code name for the code value given by *expression,* using the data dictionary domain of the element *name*. The *name* must be a data element name or its alias from an EXEC SQL statement. The value of the expression must be a number representing the internal identifier for a code. This is the value normally returned by a query. If *expression* is the same as *name* you do not have to specify it twice.
Example: if the variable *compkind* were returned from a query, the expression CODENAME(compkind) would produce a string normally displayed in NASIS for that data element, such as "series". Code names are generally in lower case. The expression CODENAME(val, compkind), where *val* is a variable from a DEFINE statement, would produce the code name for a compkind whose value is in the variable *val*.

CODELABEL (*expression* [*, name*])
Returns the code label for the code value given by *expression,* using the data dictionary domain of the element *name*. This operates just like CODENAME. The code label is typically the same as the code name but is capitalized properly for use in reports.
Example: in the above example, the expression CODELABEL(compkind) would produce "Series".

ARRAYCOUNT (*expression*)
Counts the number of non-null values in a multiple valued expression. It can operate on either a character or numeric argument, and will return a single numeric value of zero or more.
Example: if the variable A has three values, 1, 2, and NULL, the expression ARRAYCOUNT(A) would produce the result 2.

ARRAYMIN (*expression*)
Computes the minimum of the values in a multiple valued expression. It can operate on either a character or numeric argument, and will return a single value of the same type as its argument. In this case, a null value is not considered to be smaller than a non-null value. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.
Example: if the variable A has three values, 1, 2, and 3, the expression ARRAYMIN(A) would produce the result 1.

ARRAYMAX (*expression*)
Computes the maximum of the values in a multiple valued expression. It can operate on either a character or numeric argument, and will return a single value of the same type as its argument. The result is null only if all values of the array

are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.
Example: if the variable A has three values, "X", "Y", and "Z", the expression ARRAYMAX(A) would produce the result "Z".

ARRAYMEDIAN (*expression*)

Locates the median value in a multiple valued expression, by sorting the non-null values and selecting the middle one. It can operate on either a character or numeric argument, but there is a slight difference in operation between the two. When there is an even number of values there is not a single middle value, so with numeric data the median is the average of the two middle values, and with character data the median is the larger of the two. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.
Example: if the variable A has three values, "X", "Y", and "Z", the expression ARRAYMEDIAN(A) would produce the result "Y".

ARRAYMODE (*expression*)

Finds the modal value in a multiple valued expression by counting the occurrences of each distinct value and returning the value that occurs most often. In case of a tie, the smallest value is returned. It can operate on either a character or numeric argument, and will return a single value of the same type as its argument. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.
Example: if the variable A has four values, 2, 3, 5, and 3, the expression ARRAYMODE(A) would produce the result 3.

ARRAYSHIFT (*expression*, *expression*)

Shifts the values in the first argument, which is a multiple valued variable, by the number of positions specified in the second argument, which has a single value. If the second argument (call it "n") is positive, the values are shifted "up", so that the value that was in position 1 moves to position n+1, and so on until the last n values are discarded. The first n array positions are assigned a null value. If the second argument is negative, the values are shifted in the opposite direction. The result has the same data type and number of values as the first argument.
Example: if the variable A has three values, 1, 2, and 3, the expression ARRAYSHIFT(A, -1) would produce a result with three values, 2, 3, and Null.

ARRAYROT (*expression*, *expression*)

Operates like ARRAYSHIFT but performs a rotation of the values in the first argument. Values shifted off one end of the array are moved onto the other end. If the number of positions shifted is greater than the number of values, the effect is to perform more than one rotation, or a rotation modulo the dimension.
Example: if the variable A has three values, 1, 2, and 3, the expression ARRAYROT(A, -4) would produce a result with three values, 2, 3, and 1.

LOOKUP (***expression*, *expression*, *expression***)
Selects one value from an array based on an index. The first expression is the key, which must be a single value, and the second expression is the index array. If the key value is found in the index array, the value from the corresponding array position in the third expression is returned, otherwise the result is null. If there is more than one match, the result has the values from all matching rows, so it is possible for the result to have more than one value. The second and third expressions must be arrays of equal dimension. A common error is to mismatch the dimensions of these two expressions, due to differences in the way they are aggregated. The first and second expressions must have the same type, and the result will have the type of the third expression.
Example: The variable *max_thickness* has a single number, the variable *horizon_thickness* has 6 numbers, and the variable *ph_r* has 6 numbers. The expression LOOKUP (max_thickness, horizon_thickness, ph_r) would return the value of *ph_r* from the horizon whose *horizon_thickness* value matches the value of *max_thickness*.

COUNT (***expression***)
Maintains a running count of the occurrences of the expression. On each iteration of the script the value of the expression is tested for a null, and if it's not null the counter's value is increased by one.
Example: a variable defined with the value COUNT(musym) could be printed at the end of a report to show the number of mapunits read (because musym can't be null).

MIN (***expression***)
Finds the smallest value of the expression. On each iteration of the script, the value of the expression is compared to an internal counter, and replaces the counter's value if the expression is smaller. If a null value for the expression is encountered, the result of MIN becomes and remains null.
Internal counters for the MIN function cannot be reset.
Example: a variable defined with the value MIN(elevation) could be printed at the end of a report to show the minimum of elevation.

MAX (***expression***)
Finds the largest value of the expression. On each iteration of the script, the value of the expression is compared to an internal counter, and replaces the counter's value if the expression is greater. Null values are smaller than any non-null value, so the result is only null if all input values are null.
Internal counters for the MAX function cannot be reset.
Example: a variable defined with the value MAX(elevation) could be printed at the end of a report to show the maximum of elevation.

SPRINTF (***"format", expression* [, *expression*] …)**

Formats one or more expression values into a character string using the C function *sprintf* (same as the Prelude *sprintf*). The first argument is a format specification, which must have a single value, and the remaining arguments are expressions whose values are to be formatted. If any of the expressions are multiple valued, the result is also multiple valued, and its dimension is that of the expression with the largest dimension.
It is the user's responsibility to see that the number and type of the expressions correspond to the format, as there is no checking performed. Character data should use the %s formatting code, and numeric data should use the %f or %g formatting code.
Null values in the expressions produce an unusual result. The formatted value plus all characters of the format string up to the next % sign are skipped.
Example: The variable *name* has one character value, "Bob". The variable *position* has two numeric values, 10 and 12. The expression SPRINTF ("%s:%.f", name, position) will produce a result containing two character values, "Bob:10" and "Bob:12".

USER
The user name from the data dictionary.
Example: if the person running NASIS has the login name "rose", the expression USER will return a single character value, "rose".

TODAY
The current date in mm/dd/yyyy format.
Example: the result of the expression TODAY might be "07/20/1998".

`unix command`
Returns the standard output from a UNIX command. The command line may contain report variables or data elements preceded with $.
Example: if the variable *file* contains a character string which is a file name, the expression `cat $file` would result in a single character string containing the complete contents of the file.

Numeric Functions
The following function expressions operate on numeric values, and produce numeric results. If the input values are character strings they are first converted to numbers.

ARRAYSUM (*expression*)
Computes the sum of the values in a multiple valued expression. It expects a numeric argument, and will try to convert character values to numbers. It returns a single numeric value. If individual values of the array are null they are treated as zeroes. The result is null only if the array has no values. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.

Example: if the variable A has three values, 1, 2, and 3, the expression ARRAYSUM(A) would produce the result 6.

ARRAYAVG (*expression*)
Computes the average of the values in a multiple valued expression. It expects a numeric argument, and will try to convert character values to numbers. It returns a single numeric value. If individual values of the array are null they are not counted in the average. The result is null if all values are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.
Example: if the variable A has three values, 1, 2, and 3, the expression ARRAYAVG(A) would produce the result 2.

ARRAYSTDEV (*expression*)
Computes the standard deviation of the values in a multiple valued expression. It expects a numeric argument, and will try to convert character values to numbers. It returns a single numeric value. If individual values of the array are null they are not included in the computation. The result is null if all values are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.
Example: if the variable A has three values, 1, 2, and 3, the expression ARRAYSTDEV(A) would produce the result 1.

WTAVG (*expression*, *expression*)
Computes the sum of the first expression's values after multiplying each by a weighting factor, taken from the corresponding value of the second expression, then divides the result by the sum of the weights. The two expressions must be arrays of the same dimension. Individual null values are ignored in computing the average. The result is null if all the individual values are null. The result has dimension 0 if the original expressions have dimension 0, otherwise it has dimension 1.
Example: The variable *comppct_r* has 3 values (40, 30, 20) and the variable *elev_r* has three values (1000, 1200, 900). The expression
WTAVG (elevation, comppct_r) would produce the value 1044.44, which is the average of the *elevation* values, weighted by the *comp_pct* values, or
(1000*40 + 1200*30 + 900*20) / (40 + 30 + 20).

SUM (*expression*)
Computes a running total of the value of the expression. On each iteration of the script, the value of the expression is added to an internal counter. The result of the function is the value of that counter at each iteration. If a null value for the expression is encountered, the result of SUM becomes and remains null.
Internal counters for the SUM function cannot be reset. If you want to compute subtotals, use the ASSIGN statement to add the value of the expression to a defined variable rather than an internal counter. Then a conditional expression can be used to reset the variable's value to 0 at the correct time.

Example: a variable defined with the value SUM(acres) could be printed at the end of a report to show the total of acres.

AVERAGE (*expression*)
Computes a running average of the value of the expression. On each iteration of the script, the value of the expression is added to an internal counter, and the result is divided by the number of values processed. If a null value for the expression is encountered, the result of AVERAGE becomes and remains null.. Internal counters for the AVERAGE function cannot be reset.
Example: a variable defined with the value AVERAGE(elev_r) could be printed at the end of a report to show the average of elevation.

LOGN (*expression*)
Computes the natural logarithm of the expression.
Example: the expression LOGN(10) produces the value 2.302585.

LOG10 (*expression*)
Computes the base 10 logarithm of the expression.
Example: the expression LOG10(10) produces the value 1.

EXP (*expression*)
Computes the exponential (e^x) of the expression.
Example: the expression EXP(1) produces the value of *e*, 2.718282.

COS (*expression*)
Computes the cosine of the expression interpreted as an angle in radians.
Example: the expression COS(0) produces the value 1.

SIN (*expression*)
Computes the sine of the expression interpreted as an angle in radians.
Example: the expression SIN(0) produces the value 0.

TAN (*expression*)
Computes the tangent of the expression interpreted as an angle in radians.
Example: the expression TAN(0) produces the value 0.

ACOS (*expression*)
Computes the arccosine of the expression, returning an angle in radians.
Example: the expression ACOS(0) produces the value of $\pi/2$, 1.570796.

ASIN (*expression*)
Computes the arcsine of the expression, returning an angle in radians.
Example: the expression ASIN(1) produces the value of $\pi/2$, 1.570796.

ATAN (*expression*)
Computes the arctangent of the expression, returning an angle in radians.

Example: the expression ATAN(1) produces the value of π/4, 0.785398.

ATAN2 (*expression, expression*)
Computes the angular component *θ* of the polar coordinates (*r, θ*) that are equivalent to the rectangular coordinates (*x, y*) given by the two expressions. This is the same as ATAN(y / x).
Example: the expression ATAN2(5, 5) produces the value of π/2, 1.570796.

SQRT (*expression*)
Computes the square root of the expression. Returns a null value if the expression is negative.
Example: the expression SQRT(2) produces the value 1.414214.

ABS (*expression*)
Computes the absolute value of the expression.
Example: the expression ABS(-10) produces the value 10.

POW (*expression, expression*)
Computes the value of the first expression raised to the power of the second expression.
Example: the expression POW(2, 5) produces the value 32.

MOD (*expression, expression*)
Computes the remainder after dividing the first expression by the second expression.
Example: the expression MOD(5, 2) produces the value 1.

ROUND (*expression* [, *expression*])
Rounds off the value of the first expression to the number of decimal places specified by the second expression. If the second expression is not used, it is assumed to be zero, which means round off to the nearest whole number. When the second expression is a positive number, it specifies the number of places to the right of the decimal point to be preserved. If negative, it means round to the specified number of places to the left of the decimal point, as illustrated in the examples.
Examples: ROUND (15.751, 1) produces 15.8
ROUND (15.751) produces 16
ROUND (15.751, -1) produces 20

REGROUP Expression

The REGROUP expression is used to perform secondary aggregation of data. It operates a little like the AGGREGATE option in a query and can be used to perform a second level of aggregation when dealing with a complex data structure. It uses two expressions, which must be arrays of the same dimension. In the expression "REGROUP array BY array …" the second array (the "BY" array) is used as a key for grouping the values from the first array (the data array). The result is a new array whose dimension is the number of unique values in the "BY" array. The values in the result are aggregates derived from each group of rows in the data array that have the same key value.

The aggregation function determines how these aggregates are produced. The types of aggregation are the same as the query AGGREGATE option, except that NONE and UNIQUE are not applicable in REGROUP, because there can be only one value in each position of the result array. The valid aggregations types are:

SUM	Computes the sum of the values in each group.
AVERAGE	Computes the average of the values in each group.
FIRST	Select the value from the first row of the group.
LAST	Select the value from the last row of the group.
MIN	Selects the smallest of the values in each group.
MAX	Selects the largest of the values in each group.
LIST	Concatenates the values (converted to character strings if numeric) into a single string with a delimiter between each value. If a quoted string is specified after the word LIST, that string is the delimiter, otherwise a comma and space are placed between each value.

Some additional rules on the REGROUP expression are:

- The "BY" array does not have to be sorted. REGROUP will always collect together all data values for each unique key value. However, the choice of value for FIRST or LAST will be affected by the order of values in the data array.
- Nulls in the data array are ignored during aggregation except for FIRST and LAST, which preserve a null if it is the first or last value found. If all data values for some key value are null the corresponding result value will be null.
- A null in the "BY" array is a valid key value and will produce a corresponding value in the result, aggregating all null key values together.

Example: These examples use the arrays A and B as inputs:

A	B
George	4
Abe	4
Sue	5
Sam	8
Mary	8
William	8

The arrays C and D are produced by the statements:

```
DEFINE C   REGROUP A BY B AGGREGATE FIRST.
DEFINE D   REGROUP A BY B AGGREGATE LIST "-".
```

C	D
George	George-Abe
Sue	Sue
Sam	Sam-Mary-William

Syntax:
DERIVE derive_list USING property_call **.**

derive_list ⇒ variable [FROM identifier] [**,** variable [FROM identifier]] ...

property_call ⇒ [**"***site_name***"** **:**]**"***property_name***"**
[**(** argument [, argument] ... **)**]

$$\text{argument} \Rightarrow \left\{\begin{array}{l}\text{variable}\\ \text{element}\\ \text{literal}\end{array}\right\}$$

Used In:
Report, Property, Calculation

Example:
```
DERIVE thickness FROM layer_thickness
USING "NSSC_Pangaea":"LAYER THICKNESS" (0, bottom).
```

The DERIVE statement invokes a property script to produce values for one or more variables. Each name listed after the keyword DERIVE becomes a local variable in the script where it occurs. It is assigned the value of the variable in the property script whose name follows the keyword FROM. If the variable name in the property script is the same as the variable to be created in the calling script, the FROM phrase may be omitted. The property must be defined with the same base table as the calling script, and the scripts are automatically synchronized to return values for the current row of the base table.

The name of the property must be in quotes, and must match the property name in the Property table exactly, including case and punctuation. The NASIS site name is optional, but should be placed before the property name to ensure that the name is unique. Spaces before and after the colon are optional.

A list of arguments can be given after the property name, to provide values for input variables if the property script has an ACCEPT statement. The order of the arguments in the DERIVE statement must correspond to the order of the input variables in the ACCEPT statement.

The arguments can be input column names, variables, or numeric or character constants in the calling script. However, recall that DERIVE statements are always executed before DEFINE statements. If an argument is a variable which is computed in a DEFINE statement, its value will be whatever is left over from the previous input iteration, even if the DEFINE appears in the script before the DERIVE. For this reason, arguments for DERIVE should be from an ACCEPT, an EXEC SQL, or constants.

Syntax:
EXEC SQL sql-select [sort_specification] [aggregation] **.**

sql-select ⇒ SELECT [FIRST n] input_column [**,** input_column] ...
FROM table_spec [**,** table_spec] ...
[WHERE where_condition [{AND | OR} where_condition] ...]
[*Informix GROUP BY clause*] [*Informix HAVING clause*]
[{ *Informix ORDER BY clause* | *Informix INTO TEMP clause* }] ;

$$\text{input\_column} \Rightarrow \left\{ \begin{array}{l} \text{element [alias]} \\ \text{expression alias} \end{array} \right\}$$

$$\text{element} \Rightarrow \left[\left\{ \begin{array}{l} tbl\_imp\_nm \\ tbl\_nm \\ \text{alias} \end{array} \right\} . \right] \left\{ \begin{array}{l} elm\_imp\_nm \\ elm\_nm \end{array} \right\} \text{[suffix]}$$

alias ⇒ *name*

suffix ⇒ { _l | _r | _h | _s | _ls | _rs | _hs }

$$\text{table\_spec} \Rightarrow \left[\left\{ \begin{array}{l} \text{EDIT} \\ \text{REAL} \end{array} \right\} \right] \text{[OUTER]} \left\{ \begin{array}{l} tbl\_imp\_nm \\ tbl\_nm \end{array} \right\} \text{[alias]}$$

$$\text{where\_condition} \Rightarrow \left\{ \begin{array}{l} \textit{Informix}\ \text{WHERE}\ \textit{condition} \\ \text{JOIN table TO table [BY } \textit{relationship\_name}\text{]} \end{array} \right\}$$

Used In:
Report, Property, Calculation

Example:
```
EXEC SQL select areaname, legenddesc, musym, muname
from legend, mapunit, outer area
where join area to legend and join legend to mapunit;.
```

An EXEC SQL statement defines a database query that supplies input to the report engine. Any database columns or expressions listed in the SELECT clause of the query may be used as variables in the rest of the script. A script almost always has a query, unless all the needed data can be obtained from parameters and derived properties. The query normally operates on the edit tables for the current NASIS session. This means that the selected set typically determines the precise records on which to operate. The primary purpose of the EXEC SQL is to specify which data elements are necessary for the report.

The EXEC SQL statement can be thought of as an extended version of the Informix Select statement. It performs the same basic function, but has additional capabilities to make report writing easier. The extensions include:

- Use of NASIS data element names as well as Informix column names
- Simple technique to specify join conditions
- Extended sort types, such as case insensitive and symbol sort
- More powerful GROUP BY features in the AGGREGATE clause, including independent aggregation by column, and crosstab formatting.

The SQL Select statement in the EXEC SQL follows the same syntax rules as queries in the NASIS Select Manager, except that a SELECT clause is required. The SELECT clause may contain data elements or expressions, following normal SQL syntax, and each column must have a unique name. If expressions are used in the select statement, an alias must be used with the expression to provide a unique name. Besides allowing most standard SQL expressions, NASIS permits the functions CODENAME, CODELABEL, CODESEQ and CODEVAL with data elements that are stored as codes. These functions cause the query to return the name, label, sequence, or internal value for a code. If none of these functions is used the query returns the internal value.

The phrase FIRST n following the word SELECT is a new SQL feature that allows you to specify the maximum number of records to be returned from a query. The records are sorted on the columns specified in the ORDER BY clause, then up to "n" of them are used as report input.

The FROM clause specifies all the tables used in the query, and may specify aliases and outer joins according to Informix syntax. Table names used in a FROM clause must be defined in the NASIS data dictionary or in an INTO TEMP clause of a prior query. The different types of CVIR scripts may be designed to search the edit tables, the permanent tables, or both. Normally, reports search the edit tables (selected set) only, while calculations and properties search first the edit tables then the permanent tables. The keyword EDIT or REAL in the FROM clause is used to override the table search option on a table by table basis. If the keyword is used, only the edit or permanent copy of the table, respectively, will be read.

The WHERE clause, in addition to normal SQL conditions, may use the “JOIN table TO table” syntax to simplify writing join conditions. The two tables in a JOIN condition must have a relationship recorded in the data dictionary (Refer to the Relationship Report in the NASIS Metadata web page, http://nasis.nrcs.usda.gov/documents/metadata). If there is more than one relationship between the two tables, the relationship name must be used.

Subqueries are also allowed in the WHERE clause, following Informix syntax with the extensions just described. It is permissible to use a JOIN condition between a table listed in the main query and a table listed in the subquery, which is a convenient way to create a coordinated subquery. Refer to the Informix manuals or other SQL references for more information about subqueries. This is an advanced query topic.

The WHERE clause may also contain symbol references of the form $name, where name is the name of a variable defined earlier in the script or a UNIX environment variable. Typically such a variable would be defined in an ACCEPT statement or a prior query. A query with symbol references is called a parametric query. Each time the variables in the script change values, the query is re-executed. This performs more slowly than a normal query, but it is sometimes necessary in order to get more control over the records read from the database. A parametric query uses global aggregation, which is described under the aggregation clause.

Following the WHERE clause, the GROUP BY, HAVING, and ORDER BY clauses can be used with the normal Informix syntax. The INTO TEMP clause may also be used (provided the ORDER BY is not used) to direct the results of the query into a temporary database table. Subsequent queries in the same script or in subreport scripts can read from the temporary table as if it were a normal NASIS table. The column names in the temporary table are the column names (or aliases) from the SELECT clause. A query with an INTO TEMP clause should not be the only query in a report because it does not return any data that the report can use. The syntax "INTO TEMP file WITH NO LOG" should be used to conserve space in the Informix log files.

NASIS tables and data elements may be called by either the logical name or the implementation name, but must use the same name wherever referenced. A data element name can be used alone if it is unique, otherwise the table name must be given also. If an alias is used for an element in a SELECT clause, that alias must be used everywhere instead of the element name. If the element is modal, the suffix (such as _l or _h) must be included after the element name. The value from each column is converted into either a character string or a floating point number for use in later calculations. The data element type determines the conversion. Numeric data elements, such as Int, Decimal, and Float, and Code elements, are converted to floating point, and everything else, including dates, is converted to character strings.

A semicolon is required to end the SQL portion of the EXEC SQL statement. Optional sort and aggregation clauses may follow the semicolon, and the whole statement is ended with a period. If neither the sort nor aggregation is used, both a semicolon and a period are still required.

A script may contain more than one query, in order to collect data from different hierarchic paths in the database. These types of data often cannot be retrieved in a single query without creating undesirable cross products. By using separate queries and aggregating the results, the data can be “de-normalized” so that data from separate paths appear as if they were repeating groups in the base table.

EXEC SQL Statement: Sort Specification

Syntax:

sort_specification ⇒ SORT [BY] sort_key [**,** sort_key] ...

$$\text{sort\_key} \Rightarrow \begin{Bmatrix} \text{name} \\ \text{number} \end{Bmatrix} \left[\begin{Bmatrix} \text{ASC[ENDING]} \\ \text{DESC[ENDING]} \end{Bmatrix} \right] \left[\begin{Bmatrix} \text{LEX[ICAL]} \\ \text{SYM[BOL]} \\ \text{INSEN[SITIVE]} \end{Bmatrix} \right]$$

Example:

```
EXEC SQL select areaname, legenddesc, musym, muname,
mapunit.seqnum
from legend, mapunit, outer area
where join area to legend and join legend to mapunit;
SORT BY areaname, mapunit.seqnum DESC, musym SYMBOL.
```

SORT is an optional clause that may be added to a query to direct the calculation engine to sort the records. Either the ORDER BY (which causes Informix to do the sorting) or the SORT may be used, and the SORT takes precedence. The SORT clause provides more options than ORDER BY. The sort key names in the SORT clause must be data element or alias names used in the SELECT clause. The direction of sorting (ascending or descending) can be specified for each sort key, with the default being ascending. The type of sort can also be specified as lexical (like a dictionary), symbol (used for symbols containing both letters and numbers), or insensitive (ignore upper and lower case distinctions). The default sort type is the one specified in the data dictionary for the element. The sort order and type keywords may be abbreviated as shown.

The difference between SORT and ORDER BY is important when the FIRST n condition is used in the SELECT clause. ORDER BY, since it is performed by the database engine, happens before the "first n" records are selected, and SORT happens after. It could even be useful to specify different columns in ORDER BY and SORT, because the first controls which records appear and the second controls the order in which they print.

EXEC SQL Statement: Aggregation Specification

Syntax:

```
aggregation ⇒  AGGREGATE  [ ROWS  [ BY ]  identifier [ , identifier ] ... ]
               [ COLUMN  identifier [ aggregate_function ]
                   [ , identifier [ aggregate_function ] ] ... ]
               [ CROSSTAB  [ BY ]  identifier [ value_specification ]
                   [ LABELS  "string"  [ , "string"] ... ]
                   CELLS  identifier  [ , identifier ] ... ]  .
```

$$\text{identifier} \Rightarrow \begin{Bmatrix}\text{element}\\ \text{alias}\end{Bmatrix}$$

$$\text{aggregate\_function} \Rightarrow \left\{\begin{matrix}\text{SUM}\\ \text{AVERAGE}\\ \text{FIRST}\\ \text{LAST}\\ \text{MIN}\\ \text{MAX}\\ \text{LIST ["string"]}\\ \text{NONE}\\ \text{UNIQUE}\end{matrix}\right\} \text{[GLOBAL]}$$

$$\text{value\_specification} \Rightarrow \left\{\begin{matrix}\text{VALUE[S] (field\_value [, field\_value] ...)}\\ \text{INTERVAL[S] (field\_value [, field\_value] ...)}\end{matrix}\right\}$$

field_value ⇒   literal

Example:

```
EXEC SQL select musym, muname, areaname,
mapunit_area_overlap.area_overlap_acres acres
from mapunit, mapunit_area_overlap, laoverlap, area
where join area to laoverlap
and join laoverlap to mapunit_area_overlap
and join mapunit to mapunit_area_overlap;
SORT BY musym SYMBOL, areaname
AGGREGATE ROWS BY musym
COLUMN muname UNIQUE, acres SUM
CROSSTAB areaname CELLS acres.
```

The aggregation clause specifies how the input records are to be grouped, and what to do with the data in each group. The identifiers in the aggregation clause are names of columns in the query. A query without an aggregation clause uses no aggregation, meaning that rows of data are used one at a time exactly as they come from the query. The aggregation clause has somewhat different effects for a parametric query. This variation is described later, after the basic aggregation process is described.

The ROWS specification defines the input grouping. It can be used only in the first query of a report, to define which set of input columns controls the report iteration. In all other queries, iteration has to occur at the base table, so ROWS is not used. In particular, for Property and Calculation scripts, aggregation occurs at the base table, so no ROWS specification is allowed.

When ROWS is used, the query must be sorted on the columns listed after ROWS. Each unique combination of values in this set of columns starts a new iteration. Without the ROWS list, each base table record starts a new iteration. There may be one or more records returned by the query in each iteration. Multiple input values in each column are combined according to the aggregation rules, to produce single values or arrays that can be used in further calculations or report output. The behavior of row aggregation is illustrated in the following example. Suppose a query includes specifications to SORT BY musym AGGREGATE ROWS BY musym. Rows with the same musym will define the report iteration. Each group of rows with the same musym is one iteration, thus this example has eight rows, but only three iterations.

musym	compname	comppct_r
12A	Hamerly	80
12A	Vallers	15
12A	Hamre	5
26B	Windsor	90
26B	Deerfield	10
130C	Dacono	85
130C	Satanta	10
130C	Altvan	5

Aggregation rules for each column can be specified after the keyword COLUMN. The default aggregation is UNIQUE for columns that have no aggregation specified. This means that when the value in a column is the same for every row of an iteration, only one value is returned for that column. If more than one value occurs in an iteration, an array is formed to return values for the column. Each distinct, non-null value is placed in a separate position of the array. The number of positions in the array (the dimension) can vary from one iteration to another, and from one column to another within an iteration.

The aggregation function NONE is similar to UNIQUE except that it does not eliminate duplicate or null values. If there is more than one input row in an iteration, the value from each row is placed in a separate array position. For each iteration, every column with NONE aggregation will have the same dimension, and the values will be in the order of the input records.

The other aggregation functions are used to reduce multiple values for a column to a single value. The aggregations have no effect when only one record occurs in an iteration. The types of aggregation are:

SUM	Computes the sum of the column's values.
AVERAGE	Computes the average of the column's values.
FIRST	Select the value from the first record of the group (useful only if the input is sorted on this column).
LAST	Select the value from the last record of the group.
MIN	Selects the smallest of the column's values.
MAX	Selects the largest of the column's values.
LIST	Concatenates the values (converted to character strings if numeric) into a single string with a delimiter between each value. If a quoted string is specified after the word LIST, that string is the delimiter, otherwise a comma and space are placed between each value.

Given the example above, suppose the query includes specifications to AGGREGATE ROWS BY musym COLUMN compname LIST, comppct_r SUM. Since aggregation for musym is not specified, the default aggregation of UNIQUE will be applied to that column to produce the following results. Note that the values in each column have been reduced to a single valued expression for each iteration.

musym	compname	comppct_r
12A	Hamerly, Vallers, Hamre	100
26B	Windsor, Deerfield	100
130C	Dacono, Satanta, Altvan	100

The keyword GLOBAL may be used after the aggregation type for a column. This causes that particular column to be aggregated over the entire set of input data, rather than one iteration. The values for that column remain constant for the whole report. One use for global aggregation is to find data for report headings. If the first input iteration is missing some data needed in a heading, a global aggregation can find the first occurrence, or all unique occurrences of the data before the report processing actually begins. Global aggregation can be used with or without crosstabs.

The *crosstab* is a special type of aggregation that assigns values to positions in an array based on the value of a controlling column. It requires a CROSSTAB column, and one or more CELLS columns. These columns become arrays, but their dimension is determined not by the number of input rows in an iteration, but by the number of values for the crosstab. This dimension is constant for the entire query. The crosstab values are defined by the VALUES list, the INTERVALS list, or by default. The default is to use all the unique values found in the input for the crosstab column.

When doing a crosstab, for each iteration of the input, the arrays of values for the CELLS columns are first set to nulls. Then, for each input record, the value in the CROSSTAB column is examined. If it is one of the values in the VALUES list or the default list, or if it falls within one of the ranges in the INTERVALS list, its position in the list is noted. For each of the columns in the CELLS list, the value from the input record is placed in that position of the column's array.

Within an iteration, the value of the crosstab column may repeat. If so, only one value can be stored in an array position for a cell, so the cell's aggregation function is applied. If a cell has no aggregation, a data row is returned for each unique value. In each such data row, all aggregated columns will have constant values. The operation of crosstab can be illustrated using the following example data:

musym	muname	areaname	acres
10A	Alpha loam, 0 to 3	X	100
10A	Alpha loam, 0 to 3	X	200
10A	Alpha loam, 0 to 3	Y	300
10A	Alpha loam, 0 to 3	Z	400
10A	Alpha loam, 0 to 3	Z	500
10B	Alpha loam, 3 to 6	X	600
10B	Alpha loam, 3 to 6	Y	700
10B	Alpha loam, 3 to 6	Y	800

This table shows a small sample of input data from the example query above. The first case shows the results of a crosstab <u>without</u> aggregation of the crosstab cells:

AGGREGATE ROWS musym COLUMN muname UNIQUE
CROSSTAB areaname CELLS acres.

musym	muname	areaname			acres		
10A	Alpha loam, 0 to 3	X	Y	Z	100	300	400
10A	Alpha loam, 0 to 3	X	Y	Z	200		500
10B	Alpha loam, 3 to 6	X	Y	Z	600	700	
10B	Alpha loam, 3 to 6	X	Y	Z		800	

In this example the column "musym" controls row aggregation. Column "muname" has the UNIQUE aggregation, so it maintains the values that correspond to each value of "musym". Notice that if "muname" does not repeat at the same frequency as "musym", it will become an array.

The columns "areaname" and "acres" become arrays of three positions each, because the crosstab column, "areaname", has three distinct values in the input sample. The values placed in "areaname" are constant, namely the column grouping values "X", "Y", and "Z". The cell column, "acres", contains the acreage values for the corresponding position of "areaname". Because there are multiple acreage values for each area in this example, the result has two rows for each symbol.

By adding an aggregation function to the “acres” columns, the crosstab produces just one row for each iteration defined by the ROWS condition, as in the following example:

AGGREGATE ROWS musym COLUMN muname UNIQUE, acres SUM
CROSSTAB BY areaname CELLS acres

musym	muname	areaname			acres		
10A	Alpha loam, 0 to 3	X	Y	Z	300	300	900
10B	Alpha loam, 3 to 6	X	Y	Z	600	1500	

When INTERVALS are used for a crosstab, the list of field values must be numbers, in an increasing order. The number of intervals is one more than the number of values. If the intervals are specified as: CROSSTAB BY x INTERVALS (n1, n2, n3), the crosstab will place the cell data into one of 4 array positions based on the value of the variable x:

x <= n1 n1 < x <= n2 n2 < x <= n3 n3 < x

The LABELS specification can specify column headings for a report, which would otherwise be the field values for the CROSSTAB BY column. See the discussion about array specifications and column specifications under the SECTION statement for more information about formatting and printing cross tabulated data.

Note on parametric queries.
When a parametric query is used (i.e., a query containing $name references) a global aggregation is performed on all columns, and no ROWS clause is allowed. A global aggregation simply means that the whole output of the query is considered to be a single iteration. The column aggregation types are used as described above. It is not advisable to use a CROSSTAB with a parametric query, because a crosstab can generate more than one set of data for each aggregation group. Only the first such set of data would actually be available in the CVIR script.

A parametric query performs a global aggregation even if no aggregation clause is used. In this case, the default column aggregation rule is NONE.

Syntax:
FONT *"font name"* .

Used In:
Report

Example:
```
FONT "-hp-line printer-medium-r-normal--*-85-*-*-*-*-iso8859-1".
```

Defines the font selected for report output. This must be a fixed-width (not proportional) font, which greatly limits the number of fonts available. Standard NASIS reports use only two fonts, to provide compatibility with the greatest possible number of printers. Your printer may provide more choices.

The FONT specification is always used with the PITCH specification to define exactly how many characters and lines per inch are printed in the selected font. The default font is Courier 12 point, which prints 10 characters per inch horizontally, and 6 lines per inch vertically. The font in the above example is a condensed font available on HP laser printers. Its pitch values are Horizontal 17 and Vertical 8 (when using a Postscript printer, a font of similar dimensions is automatically substituted). Another font available on most printers is intermediate between the "line printer" and the default font. This is a 10 point Courier, which would be specified as:

```
FONT "-adobe-courier-medium-r-normal--*-100-*-*-*-*-iso8859-1".
PITCH HORIZONTAL 12 VERTICAL 7.
```

Syntax:

```
HEADER  [ INITIAL ]
        line-specification ...
END HEADER .

FOOTER  [ FINAL ]
        line-specification ...
END FOOTER .
```

Used In:

Report

Example:

```
HEADER
AT CENTER "Sample Report".
SKIP 2 LINES.
END HEADER.
```

Defines the headers and footers for the report. There are four types of header/footer statements, and a report may contain no more than one of each type. All are optional. The default for HEADER and FOOTER is to print nothing. The default for HEADER INITIAL or FOOTER FINAL is to print the HEADER or FOOTER, respectively.

The regular header and footer are printed at the top and bottom, respectively, of each report page. The initial header and final footer are printed only once, at the beginning and end of the report instead of the regular header and footer. At the end of the report, if there is not enough room for the final footer (which could happen if the final footer uses more lines than the regular footer), the regular footer is printed on the last page of data, then the final footer is printed on a separate page. Each header or footer contains one or more line specifications as defined below (except for NEW PAGE commands). The text of headers and footers is generated one time, at the beginning of report execution, and reprinted at the top of each page. Page numbers, if included in headers and footers, will be substituted correctly. Data from the database used in headers or footers will come from the first input record only.

Syntax:
INPUT input-list FILE filename [DELIMITER **"***string***"**] [sort-specification]
[aggregation] **.**

input-list ⇒ input-column [**,** input-column] ...

input-column ⇒ *name* [CHARACTER | NUMERIC] [alias]

$$\text{filename} \Rightarrow \begin{Bmatrix} \text{parameter\_name} \\ \textbf{"}\textit{string}\textbf{"} \\ \textbf{\$}\textit{environment\_variable} \end{Bmatrix} \left[/ \begin{Bmatrix} \text{parameter\_name} \\ \textbf{"}\textit{string}\textbf{"} \\ \textbf{\$}\textit{environment\_variable} \end{Bmatrix} \right] \ldots$$

Used In:
Report, Property, Calculation

The INPUT statement reads data from a file into CVIR variables. Each column name in the input-list (or alias if used) becomes a variable in the script. If the column name is a NASIS data element name the data type for the column is the same as the element's. If not, either CHARACTER or NUMERIC must be specified.

The file name is composed of segments taken from environment variables, parameters (of CHARACTER type only), or quoted strings. The segments are separated by slash marks as in UNIX file names. A quoted string can also contain slashes. Examples are:

```
INPUT col1, col2 FILE $NASISLIB/"lookup.data".

#PARAMETER filename CHARACTER PROMPT "Data File Name".
INPUT col1, col2 FILE $HOME/filename.

INPUT areaname, areaacres FILE "/usr/tmp/myfile".
```

Note that the variable $NASISLIB is set when NASIS is started and refers to the directory ~nasis/lib. This could be a convenient place to store data files used by reports that are for general use; contact the Soils Hotline about placing files in this directory. An INPUT statement like the first example above would be used to access this file.

Input from a file can be aggregated, as described above for queries, to produce single or multiple valued variables. If the INPUT statement precedes any queries in a script, the report will have an iteration for each input record just as if a query were used. A BASE TABLE declaration cannot be used in this case. If the INPUT appears after a query, the aggregation for the INPUT is assumed to be global, similar to a parameterized query.

The input record must be in ASCII character format. A delimiter follows each data value in the input record. Any character string can be specified as the delimiter. The default is "|", as used in Informix unload files.

Syntax:

INTERPRET rule [, rule] … [BY *table_name*] [MAX *n* REASONS]
    [DEBUG column-list]

  rule ⇒ [**“***site_name***” :**]**“***rule_name***”**

  column-list ⇒ *element_name* [**,** *element_name*] ...

Used In:

Report

Generates interpretations for inclusion in a report. One or more rules can be specified, and the interpretation values will be computed based on these rules for each record in current selected set. These values are written to a temporary database table, which can then be queried in an EXEC SQL statement. The interpretations are produced for the database table specified after the word BY. If BY is not specified the report’s BASE TABLE is assumed. Either a BY or a BASE TABLE statement must be used when INTERPRET is used, and the table must be the same as the base table for the properties used in the interpretation.

The rule list specifies rules whose values will be generated. Each rule name is written as “Site Name”: “Rule Name”. Site Name is the NASIS site that owns the rule, and Rule Name is the name of the rule. Each name must be in quotes. This can be written as just “Rule Name” provided that the rule name is unique.

The optional phrase MAX n REASONS can be used to limit the number of reasons (sub-rules) whose results will be returned from the interpretation. Regardless, all sub-rules are used to derive the interpretation results; this only limits how many are returned to the temporary table (*i_table*) for use by the report script. If *n* is zero or this phrase is omitted, all sub-rule results will be included, even if their values are zero. If *n* is greater than zero, the highest *n* non-zero sub-rule results will be returned. The sub-rules are always sorted so the highest values are first.

The optional DEBUG phrase is used to produce debugging information from the interpretation generator. This lists the results of every property, evaluation and sub-rule used, and can be very voluminous. It is only used when writing and testing new rules. Because of the size of the debug output, only a few components should be in the selected set. The column-list is a comma separated list of names of columns to be printed at the start of each debug list. These must be columns in the interpretation base table. For typical interpretations, the base table is component, and the debug columns would probably be data_mapunit_iid_ref and component_name. These would print for each component on the debug listing to identify the component.

Examples:

```
INTERPRET  “ENG - Shallow Excavations” BY component.
```

Generates one interpretation for each component in the selected set, and returns its results along with all sub-rule results.

```
INTERPRET  "NSSC_Pangaea": "FOR-Harvest Equipment Operability",
     "NSSC_Pangaea": "FOR-Log Landing Suitability"
     MAX 5 REASONS.
```

Generates results for two interpretations and returns up to 5 sub-rule results for each rule. Only non-zero sub-rule results will be returned. Notice that the rule names are fully qualified by NASIS site name.

```
INTERPRET "ITC_Prototype":"Test Rule"  DEBUG dmuiidref, compname.
```

Runs an interpretation for testing with debugging output.

Using Interpretations in Reports:

The results of the INTERPRET command are placed in a temporary database table named either *i_component*, *i_datamapunit* or *i_pedon* depending on the base table for the interpretation. In principle, any of these tables could be the base table of interpretations, but in most cases it will be the component table. To use a different base table, all properties must use that base table, which means there must be a whole set of evaluations and rules to go with them. The following will focus on component as the base table, but the discussion is the same for other *i_tables*.

The columns of the *i_component* table are:

Element name	Column name	Description
component_iid_ref	coiidref	Component id used to link to the component table.
rule_iid_ref	ruleiidref	Rule id of the rule.
rule_name	rulename	Name of the rule.
interp_low_low	interpll	The fuzzy value of the minimum rating for the rule.
Interp_low_low_class	interpllc	The rating class name of the minimum rating.
interp_low_rv	interplr	The fuzzy value of the minimum of the representative values of the ratings.
interp_low_rv_class	interplrc	The rating class name of the minimum of the representative values of the ratings.
interp_high_rv	interphr	The fuzzy value of the maximum of the representative values of the ratings.
interp_high_rv_class	interphrc	The rating class name of the maximum of the representative values of the ratings.
interp_high_high	interphh	The fuzzy value of the maximum rating for the rule.

interp_high_high_class	interphhc	The rating class name of the maximum rating.
interp_flags	interpflags	Flags indicating use of null or default data in evaluation. The flag values can be: 1: Null data present in input 2: Default value used for a property 4: Inconsistent data detected If more than one of these conditions is present the values are added together.
sequence_number	seqnum	The sort sequence of the ratings for a top level rule.
main_rule_iid_ref	mruleiidref	The rule id of the top level rule.
main_rule_name	mrulename	The name of the top level rule.
rule_depth	ruledepth	An indicator of the depth of the rating, where 0 is the top level.

The rule id columns typically are not needed in a report, since it is more useful to print the name of the rule. Rating values can be printed either as fuzzy values (numbers between 0 and 1) or as rating class names, or both. To produce results in the order intended, the data should be sorted first on component, then on main rule name, then on sequence number. Sorting on sequence number is important because the interpretation engine assigns sequence numbers so that each subrule will come out after its parent rule, with the highest rating values first. Rule depth can be used with the NEST option (described in column layout specifications under the SECTION statement) to print subrules indented below their parent rules. The rule depth can also be used in the Where clause of the query to select only ratings for the top level rule (where rule_depth = 0) or, for example, the top rule and its first level reasons (where rule_depth < 2).

The table shown below illustrates interpretive data produced in the i_component table for one interpretation on one component. Internal record identifiers are not included in this example. Note that sorting on seqnum places the subrules (ruledepth = 1) in the correct order under the main interpretive rule (ruledepth = 0). Data in the i_component table (and other i_tables) can be queried in the same manner as any other table in NASIS, but i_tables are temporary and discarded when the report is completed.

```
compname|mrulename   |seqnum|rulename    |ruledepth|interpll|interplr|interphr|interphh|interpllc|interplrc|interphrc|interphhc|flags
________|____________|______|____________|_________|________|________|________|________|_________|_________|_________|_________|_____
WINDSOR |AWM - Land A|     0|AWM - Land A|        0|    1.00|    1.00|    1.00|    1.00|Very limi|Very limi|Very limi|Very limi|    3
WINDSOR |AWM - Land A|     1|Filter Field|        1|    1.00|    1.00|    1.00|    1.00|Filtering|Filtering|Filtering|Filtering|    0
WINDSOR |AWM - Land A|     2|Adsorption C|        1|    1.00|    1.00|    1.00|    1.00|Low adsor|Low adsor|Low adsor|Low adsor|    3
WINDSOR |AWM - Land A|     3|Surface Reac|        1|    0.21|    0.91|    0.91|    1.00|Too acid |Too acid |Too acid |Too acid |    0
WINDSOR |AWM - Land A|     4|Droughty, AW|        1|    0.00|    0.26|    0.26|    0.98|Not droug|Droughty |Droughty |Droughty |    0
```

Syntax:
MARGIN [LEFT *number* [IN]] [RIGHT *number* [IN]]
[TOP *number* [IN]] [BOTTOM *number* [IN]] **.**

Used In:
Report

Example:
```
MARGIN TOP 1 inch BOTTOM 1 inch.
```

Defines margins for the report pages. Defaults are one half inch for all margins. If margins are specified with IN, INCH, or INCHES, they are measured in inches, otherwise they are in lines or characters. The relationship between lines, characters and inches is defined by the PITCH specification.

When report output is saved to a file as text, all margins are removed. This allows a report to be viewed or printed with margins added for legibility, while removing unneeded blank space in the text copy. Removing margins makes it easier to import the text copy into a word processor or other program.

Syntax:

PAGE [LENGTH { *number* [IN] | UNLIMITED }]
[WIDTH { *number* [IN] | UNLIMITED }] **.**

PAGE PAD
line-specification ...
END PAGE PAD .

Used In:

Report

Example:

```
PAGE WIDTH 144 LENGTH 88.

PAGE PAD
  USING normal_template.
END PAGE PAD.
```

The Page Length/Width statement defines the size of the physical page for report output. The default size is length 11 inches and width 8.5 inches. If sizes are specified with IN, INCH, or INCHES, they are measured in inches, otherwise they are in lines or characters. The relationship between lines, characters and inches is defined by the PITCH specification.

Length can be specified as UNLIMITED, which means that the whole report is treated as a single page. This can be used in reports that are for screen display or for saving as text, but if the output is sent to a printer only one page is printed, containing only as much output as fits on the page. Also, some lines might not print because there are no margins at the top and bottom of pages. .

Width can also be specified as UNLIMITED, which means that report lines are as long as the data in them requires. This is useful when the report output is saved as text, but sending the output to a printer would probably not be desirable.

When a report preview is requested with unlimited length or width, the preview window cuts off the output at 100 lines long and 100 characters wide.

Page padding is used when lines are skipped at the end of a page, or when the FILL or NEW PAGE commands are used. The line specification in the PAGE PAD block is printed instead of blank lines. If the block contains more than one line, the whole block of lines is printed repeatedly to fill the required space. Page padding can be useful in subreports when you want to be sure that a complete page is returned to the calling report. If no page padding is used the size of each subreport page could vary, because KEEP blocks may force a new page to start before the bottom of the previous page is reached.

Syntax:
#PARAMETER name [parameter_attribute] ... .

$$\text{parameter\_attribute} \Rightarrow \left\{ \begin{array}{l} \text{ELEMENT element} \\ \text{PROMPT "}\textit{string}\text{"} \\ \text{MULTIPLE} \\ \text{SEARCH} \\ \text{SELECTED} \\ \text{value\_type} \end{array} \right\}$$

$$\text{value\_type} \Rightarrow \left\{ \begin{array}{l} \text{CHARACTER} \\ \text{NUMERIC} \\ \text{BOOLEAN} \\ \text{CODEVAL} \\ \text{CODESEQ} \\ \text{CODENAME} \\ \text{OBJECT} \end{array} \right\}$$

Used In:
Report

A PARAMETER allows the user to customize the report script through a dialog. Parameter references can generally be used wherever a variable, a literal, or a list of variables or literals can be used. When a parameter name appears in a statement or expression, its value is substituted like a macro. Parameters are commonly used in WHERE clauses, or to provide a rule name in the INTERPRET statement, or for column names in a CROSSTAB.

The PARAMETER definition statement is normally placed at the beginning of the report script and is interpreted by a special scan prior to processing the whole report. The statement begins with a # symbol in the first column of a line, which is normally treated as a comment in a report script. Following the parameter name, one or more attributes may be specified to create a more meaningful parameter dialog. The attributes are:

> ELEMENT means that the parameter takes the same values as the named data element. If the element has a choice list, that list appears in the parameter dialog. The element's data type will also apply to the parameter value(s). The element name should be written as *tbl_nm.elm_nm* to be sure that the name is unique.
>
> PROMPT provides a label for the input field in the parameter dialog. This can give the user hints on how to fill in the parameter value. If no prompt is provided,

the Label field from the ELEMENT definition is used as the prompt. If neither PROMPT nor ELEMENT is provided, the parameter name is used as the prompt.

MULTIPLE means that more than one value can be entered for the parameter. If a choice list is used, multiple choices can be selected. The result of the parameter dialog is formatted as a character string with commas separating the values. If the values are of character type, each individual value is surrounded by quotes.

SEARCH means that a choice list will be built for use in the parameter dialog by searching the database for all unique values entered for the specified data element. The ELEMENT attribute must be used with SEARCH. Note that it could take some time to build a choice list if the element is in a table with many rows.

SELECTED is like SEARCH but only searches the data in the current selected set. This will present the user with a list of choices that could actually appear in the report. An example is a choice list for crop name. Normally this would list all crop names in the domain, but with SELECTED the choice list only includes crops that actually occur in the selected legends.

The value type option allows the type of the parameter to be specified when the ELEMENT attribute is not used, or when code conversions are needed. The type tells the parameter dialog how to format the parameter value. Only one value type may be specified. The allowed types are:

CHARACTER means that the value entered by the user will be surrounded by quotes when it appears in the report script. This is the default if neither type nor ELEMENT is specified.

NUMERIC means that the user must enter a number, and the quotes are not used.

BOOLEAN means that the parameter dialog will display a toggle button instead of a data entry field. The parameter's value is numeric, and contains a 1 or zero indicating whether or not the user toggled the button.

CODEVAL can be used when the parameter refers to a data element that uses codes. This option specifies that the parameter value will be returned as the code's value, although the choice list contains code names. The default for coded elements is CODEVAL.

CODESEQ can be used when the parameter refers to a data element that uses codes. The parameter value will be returned as the code's sequence number.

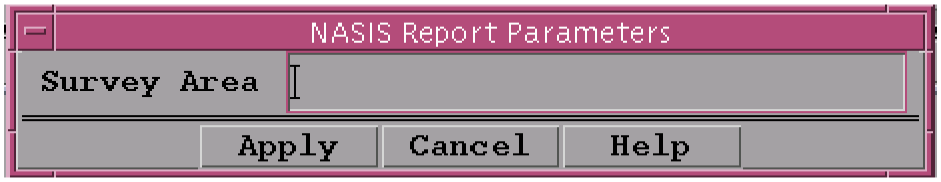
CODENAME can be used when the parameter refers to a data element that uses codes. The parameter value will be returned as the code's name.

OBJECT means that the parameter is an object name, such as a rule or property name. The parameter dialog displays an entry field for a NASIS site and a choice list for object names. The parameter must refer to an element in the root table of a NASIS object, typically the name column. The value returned is in the format used in the DERIVE and INTERPRET statements, namely “site” : “name”. If used with the MULTIPLE option, a comma separated list of object names is returned.

The following fragments of report scripts illustrate the use of parameters:

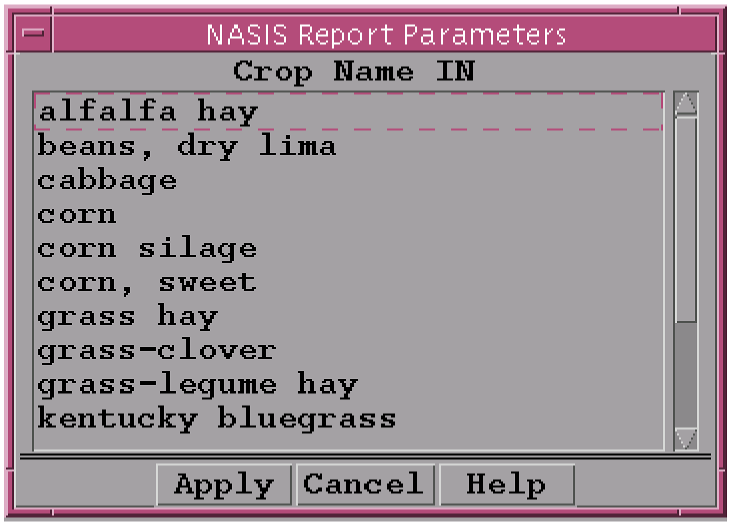
```
#PARAMETER aname ELEMENT area.areaname PROMPT “Survey Area”.
...
EXEC SQL select ... where area.areaname = aname and ...
```

This asks the user to provide a survey area name, which is then used in a query to get records for the selected area. The parameter dialog would look like:



```
#PARAMETER crops ELEMENT dmucropyld.cropname MULTIPLE SELECTED.
...
... CROSSTAB BY dmucropyld.cropname VALUES crops
```

This example allows the user to select one or more crop names from a choice list based on the contents of the selected set. The names will be used as column headings in a crop yield report. The prompt will be the label for the element *dmucropyld.cropname*, which is “Crop Name”, as shown:



Syntax:
PITCH [HORIZONTAL *number*] [VERTICAL *number*] **.**

Defines the character spacing, in characters or lines per inch. The default is horizontal 10 characters per inch and vertical 6 lines per inch. The pitch specification must match the dimensions of the font specified in the FONT statement or the report output will not fit the page correctly. In the current implementation the writer of the report script must determine the appropriate pitch because the program is not able to get this information from the font specification. See the FONT statement for further details.

Used In:
Report

Example:
```
PITCH HORIZONTAL 17 VERTICAL 8.
```

Syntax:
SECTION [section-name] [keep-option] [condition]
[HEADING line-specification ...]
[DATA line-specification ...]
END SECTION **.**

section-name ⇒ *name*

Used In:
Report

Example:
```
SECTION WHEN LAST OF musym KEEP WITH main
DATA    AT 40 "----------".
    AT 40 total_acres width 10 decimal 2.
END SECTION.
```

A report section defines a collection of one or more contiguous lines of the report output. A section can be unconditional, meaning that the section's data lines are printed on each iteration of the report's main query, or they can be printed only when certain conditions occur. A report can have any number of sections, which are evaluated in the order they appear in the report script.

To take a simple example, imagine a report script having a section "A", which prints the mapunit symbol and mapunit name, followed by a section "B" that prints the component name. Section B is unconditional, and section A prints whenever the value of the variable "musym" changes. This would be defined in the following manner:

```
SECTION A WHEN FIRST OF musym
DATA
    AT LEFT musym, muname.
END SECTION.

SECTION B
DATA
    AT LEFT compname.
END SECTION.
```

The output of the report might be:

```
12A Hamerly-Vallers complex, 0 to 2 percent slopes
Hamerly
Vallers
Hamre
26A Windsor loamy sand, 0 to 3 percent slopes
Windsor
```

This would be produced from a query returning 4 records for the two mapunits. The first mapunit has three components and the second mapunit has one component. Since section A appears in the report script before section B, and the first value for "musym" is considered a change of value, the content of section A is printed first. Then section B is printed for each input record until a change in musym occurs. Then section A is printed again, and finally section B is printed for the last record.

To define a section, specify one or more of the following features, each of which is discussed in more detail later.

1. A section can be given a name. Names are used in the KEEP option, and can be useful as documentation.
2. A KEEP controls the splitting of the section when the end of a page is reached.
3. A condition specifies when the section is used. If no condition is provided, the section appears for each report iteration.
4. If a heading block is provided, it prints at the top of the report page after the general header. If the section has no condition, the heading prints on every page, but if the section has a condition the heading only prints if the condition is true when it is time to start a page. The heading block contains one or more line specifications. If any data element values are printed in a heading, they will come from the record being processed at the time the heading prints (Note that this differs from the use of data in headers and footers).
5. If a data block is provided, it prints on each report iteration for which the condition holds. The data block contains one or more line specifications. All the lines for the data block print when the section prints, but array values or text wrapping can result in some lines repeating for a single section instance.

SECTION Conditions

Syntax:

$$\text{condition} \Rightarrow \text{WHEN} \left\{ \begin{array}{l} \text{boolean\_expression} \\ \text{break\_condition} \\ \text{AT START} \\ \text{AT END} \\ \text{NO DATA} \end{array} \right\}$$

$$\text{break\_condition} \Rightarrow \left\{ \begin{array}{l} \text{FIRST} \\ \text{LAST} \end{array} \right\} \text{[OF] identifier [, identifier]}$$

Example:

```
SECTION WHEN type == 2
```

A condition can be an ordinary boolean expression based on data from the database or internal variables. In this case, the section prints whenever the condition evaluates to True. Boolean expressions are described under the DEFINE statement.

Another form of the condition detects control breaks in the report data. This type of condition begins with the keyword FIRST or LAST. At least one of the identifiers in the break condition should be a data element in the sort key for the main report query. A control break occurs when the value of any specified element, or of any element higher in the sort key, changes. The choice of FIRST or LAST in the break condition determines which data are used for the lines printed in the section. With FIRST, the first record with the new value of the control variable is used, while LAST uses the last record with the old value. The LAST condition would be used for printing subtotals for a group of records, while FIRST would be used for printing a heading line before a group of records.

The remaining conditions are used for special conditions that occur no more than once in a report.

The AT START condition means that the section prints before any other sections (but after the headers), while an AT END section prints after the last data record (but before the footers). The default for these sections is no printing.

A NO DATA section prints only if there are no input records, and could be used print a message such as “No data found”. If the NO DATA section is not used and there is no input, no report output is produced. Instead, a warning dialog is displayed to the user.

The operation of headings in conditional sections, can be a little unexpected. When a heading block is specified in an unconditional section, the result is simple. The heading lines print on each report page following the page header. The headings appear in the order that the sections are defined. To reduce confusion, it is a good idea to include all unconditional headings in a single section, and place this section first in the script. In a simple report, both headings and data can be specified in the same unconditional section.

If a conditional section has a heading, the heading only prints if a page break occurs while the conditional section is being printed. It helps to arrange for a page break to occur just before printing the conditional section. This feature can require some trial and error to get the desired results.

Heading lines can contain references to data elements or variables, whose values print in the heading. Note that headings are generated each time a new page begins, so the heading will contain values in effect at the time they print. In particular, a LAST OF section will use values from the last record before the control break, and a FIRST OF section will use values from the new record (the one causing the control break). Note however, if a LAST OF section (or any other type of section) causes a page break, all the headings on the new page will use data from the new record.

SECTION KEEP option

Syntax:
keep-option ⇒ { NO KEEP | KEEP WITH section-name [, section_name] ... }

Example:
```
SECTION b KEEP WITH A
```

The KEEP option controls what happens when the end of a page is reached while a section is being printed. Without any KEEP option, the default behavior is to allow a page break to occur after printing all the lines defined for one section occurrence. If the DATA block contains more than one line specification, or if continuation lines are needed for long text fields, these output lines will be kept together on a page. The NO KEEP overrides this by allowing page breaks between lines of a section, although text continuation is still kept on a page if possible.

The KEEP WITH option specifies other sections with which this section is linked. This means that when a section immediately follows an occurrence of one of its "keep with" sections, the data block for the new section occurrence must fit on the same page as the last line of data in the "keep with" section. If there is not room, a page break is inserted before the last keep block of the named section.

Line Specifications

Syntax:

line-specification ⇒ [IF expression] line-content

$$\text{line\_content} \Rightarrow \left\{ \begin{array}{l} \text{SKIP } \textit{number } \{\text{LINES} \mid \text{INCHES}\}. \\ \text{FILL } \textit{number } \{\text{LINES} \mid \text{INCHES}\}. \\ \text{NEW PAGE}. \\ \text{INCLUDE subreport [(argument [, argument] ...)]} \\ \text{USING template\_name column\_spec [, column\_spec] ... .} \\ \text{AT position [alignment] column\_spec [, column\_spec] ...} \\ \quad \text{[; AT position [alignment] column\_spec [, column\_spec] ...] ... .} \end{array} \right\}$$

subreport ⇒ [**“***site_name***”** **:**]**“***report_name***”**

$$\text{argument} \Rightarrow \left\{ \begin{array}{l} \text{variable} \\ \text{element} \\ \text{literal} \end{array} \right\}$$

position ⇒ { *number* [IN] | LEFT | RIGHT | CENTER }

alignment ⇒ { TOP | BOTTOM | SAME }

Examples:

```
SKIP 2 LINES.
AT LEFT musym WIDTH 8, muname WIDTH 50.
IF comp_pct > 10 USING comp_tmpl compname, slope_l, slope_h.
AT 5 name WIDTH 20; AT 24 BOTTOM ARRAY (acres width 8 decimal 0).
INCLUDE "MLRA10_Office":"Flood Subreport" (dmudbsidref, coiid).
```

A line specification is used either to control spacing on the page or to produce actual report output. Line specifications can be either conditional or unconditional. When the IF clause is used, the IF expression is evaluated each time the section is processed. The expression follows the same rules as expressions for the DEFINE statement (see page 6). If it results in a True (non-zero) value, the line is printed. If the value of the expression is False (a null, a zero or an empty character string) nothing is printed. Without the IF clause, the line is unconditional and is always printed when its section is printed.

The line in this context is sometimes called a “logical” line because it is a single unit of output, even though it may include several “physical” lines on the report page. For example, if a logical line contains a text field it may require several lines on the page to print all the text. According to the KEEP rules a whole logical line is always kept together on one page, unless the text requires more than a full page to print.

The line_content portion of the command determines the content of the line:

1. The SKIP command produces the specified amount of blank space. If the bottom of a page is reached, the NEW PAGE action is taken, and any remaining skip lines are discarded. Either LINES or INCHES must be specified for the amount to be skipped.

2. The FILL command is like the SKIP command, but it fills the specified space with repetitions of the PAGE PAD block.

3. The NEW PAGE fills out the page with repetitions of the PAGE PAD, then prints the footer, starts a new page, and prints the header. If a NEW PAGE occurs at the very end of a report, the report generator will ignore it and not print an extra blank page.

4. The INCLUDE command runs another report and inserts its output as a logical line in the first report. Arguments may be passed to the subreport, and they must correspond with variables in the subreport's ACCEPT statement. Typically a record key would be passed as an argument, which would be used by the subreport to query for information related to that record. A report and its subreports do not need to use the same base table, and no automatic synchronization is done as with properties in a DERIVE statement. Subreports may call themselves in a recursive fashion to produce a report on recursively organized data. An example is a report to list rules and all their subrules at any depth.

   Tips: Because the entire output of a subreport is inserted as a single logical line, it is advisable to design subreports so their output is less than a page and they don't use page footers. Longer output will spill over onto additional pages of the main report and possibly produce unwanted results. However, it is also possible to have a main report that produces no output of its own and only calls a series of subreports, in which case the main report will be a page by page copy of the subreports. A main report and its subreports should use the same page and font sizes.

5. The USING statement specifies a template to serve as a format for the line. The column specifications in the USING statement are matched to the FIELD keywords in the template. The element or variable specified in the column spec is printed with the formatting defined in the template. But any formatting options specified in the USING override the corresponding options in the template. If the USING does not have as many columns as there are FIELDs in the template, the remaining fields are printed as blank. The USING may not have more columns than the template has FIELDs. Columns in the USING statement may not use the ARRAY or FIELD options.

6. A report line may be defined by specifying one or more columns with explicit positions, by use of the AT keyword. Only this form of the line specification can be used in the TEMPLATE command. The column position can be a number, expressed in characters or inches from the left margin, or it can be the left, right, or center of the line, relative to the margins. The column position may be followed by an alignment, which defines where this group of columns appears relative to the previous AT group, as shown in the example below. If no alignment is specified the default is TOP.

   Following the position, one or more columns are specified. These columns print adjacent to each other in order from left to right, until a new AT keyword and position is reached. A semicolon must separate AT groups, as shown in the syntax. Note that when more than one column spec follows an AT RIGHT or AT CENTER, the group of columns is first strung together, then right justified or centered as a unit.

   The columns in an AT group have another relation to each other when printing data from array variables. Within a group, values printed on the same line always come from the corresponding positions of each array variable. If data in one column wraps, the other columns will be blanked as needed to maintain alignment. For columns in different AT groups, word wrapping can cause data from different array positions to appear on the same line (which is desirable in some reports). The following example illustrates this:

   This line specification uses columns “name”, “age” and “score” from a query with aggregation type NONE, so each column contains an array of values. The column “text” comes from a different query and has only one value, which is a long text string.

   ```
   AT LEFT name width 12, age width 6, score width 7; AT 28 text width 28.
   ```

   This could produce the following output. Note that “text” wraps across several lines, and is not associated with any one of the name lines. But when a name wraps, the associated data stays in alignment.

   ```
   Jones           30    5.9  This group of people has
   Abercrombie-    52    5.4  responded to all of the
   Fitch                      surveys conducted since May,
   Smith           27    6.1  1983.
   Martinez        41    5.7
   ```

   The line with the name “Abercrombie-Fitch” requires two lines of output because the name doesn’t fit in 12 characters. The age and score are printed on the first of these lines, and a blank appears beneath them due to wrapping in the first column.

   In some cases this might not be quite the desired output. If you want the age and score to appear lined up with the end of the name rather than the beginning, it would require use of the alignment option. Age and score would have to be in a separate AT group using BOTTOM alignment, meaning that they line up with the bottom of the previous AT group. The following example shows this:

```
AT LEFT name width 12; AT 13 BOTTOM age width 6, score width 7; AT 28 text width 28.
```

This version would produce the following output.

```
Jones           30    5.9  This group of people has
Abercrombie-                responded to all of the
Fitch           52    5.4  surveys conducted since May,
Smith           27    6.1  1983.
Martinez        41    5.7
```

There is a third possible alignment option, SAME. This is used in cases where there are three or more AT groups, the second one has BOTTOM alignment, and both of the first two could have wrapping of long text. Then there are three possible places for the third AT group to line up: the original top line for the record, the bottom of the first group (which is the same as the second group) or the bottom of the second group. These three positions correspond to the alignments TOP, SAME, and BOTTOM. In NASIS the national manuscript report Table E2 uses this feature, if you want to see an example.

Column Specifications

Syntax:

$$\text{column\_spec} \Rightarrow \left\{ \begin{array}{l} \text{literal} \\ \text{identifier} \\ \text{FIELD} \\ \text{PAGE} \\ \text{PAGES} \\ \text{array\_spec} \end{array} \right\} [\text{ column\_layout }]$$

array-spec $\Rightarrow$ ARRAY **(** column-spec [**,** column-spec] ... **)**

The column specification identifies exactly what will be printed at a particular spot in a report. A column can print data from a literal, variable, data element, or page number. It can also be a compound column (ARRAY) consisting of one or more sub-columns. In a template definition, the keyword FIELD is used as a place holder, with the actual element, variable, or literal to be supplied later.

If a variable or element is printed, its value at each report iteration prints according to the layout options. If a literal is used, it prints the same value each time. The keywords PAGE and PAGES generate page numbering, and are normally used in headers or footers. Wherever the word PAGE occurs, the number of the current page is substituted, before column layout options are applied. The keyword PAGES is replaced by the total number of pages in the report, as in "Page n of m".

When the ARRAY specification is used, a group of one or more columns is printed repetitively, with the same format. Array columns are used only with crosstab reports. The number of columns actually printed equals the number of column values in the crosstab, times the number of column specs in the ARRAY spec. The printing sequence is to print all the columns listed in the ARRAY spec, then repeat for the number of crosstab values. Any column layout options listed outside the parentheses of an ARRAY spec apply to all columns within the parentheses, unless overridden by layout options inside the parentheses which apply to an individual column.

In the description of the EXEC SQL Statement, there is an example of a crosstab on page 27. It produced the data shown in the following table. The variables *areaname* and *acres* are arrays with 3 values each.

musym	muname	areaname			acres		
10A	Alpha loam, 0 to 3	X	Y	Z	100	300	400
10A	Alpha loam, 0 to 3	X	Y	Z	200		500
10B	Alpha loam, 3 to 6	X	Y	Z	600	700	
10B	Alpha loam, 3 to 6	X	Y	Z		800	

The following example shows one way these data could be printed. Ignoring column formatting details for the moment, these line specifications for heading and data would produce the report fragment shown.

```
HEADING
AT 1 musym LABEL, muname LABEL, ARRAY(areaname).
DATA
AT 1 musym, muname, ARRAY(acres, “acres”).
```

musym	muname	X	Y	Z
10A	Alpha loam, 0 to 3	100 acres	300 acres	400 acres
10A	Alpha loam, 0 to 3	200 acres	acres	500 acres
10B	Alpha loam, 3 to 6	600 acres	700 acres	acres
10B	Alpha loam, 3 to 6	acres	800 acres	acres

The heading line prints the labels for the data elements *musym* and *muname*, which we assume are just the column names, then the values for *areaname*, which define the groupings.

The data line prints *musym* and *muname*, this time as normal report columns, then *acres* and the literal “acres” as an array. The values from *acres* are paired with the word “acres” and printed in three columns. In this example, the crosstab was not set up with aggregation, so there are several blank spaces, but the literal prints anyway. The report could be made to look better by changing the crosstab, or moving the word “acres” into the heading.

When a multiple valued variable is printed in a column that does not have an array spec, the values are printed one beneath the other in the column. It results in a set of parallel report columns for each query column, as illustrated earlier.

Column Layout Specifications

Syntax:

```
column-layout ⇒  [ WIDTH number [IN] ]
                 [ WIDTH UNLIMITED ]
                 [ LABEL ]
                 [ DIGITS number ]
                 [ DECIMAL number ]
                 [ SIGDIG number ]
                 [ ALIGN { LEFT  |  CENTER  |  RIGHT  } ]
                 [ PAD "character" ]
                 [ INDENT number [IN] ]
                 [ NEST number [IN] PER identifier ]
                 [ NO COMMA ]
                 [ TRUNCATE ]
                 [ REPEAT ]
                 [ SEPARATOR "string" ]
                 [ REPLACE NULL [WITH] literal ]
                 [ REPLACE ZERO [WITH] literal ]
                 [ SUPPRESS [DUPLICATES] [ BY identifier ] ]
                 [ QUOTE[D]  [ quote-string ]  [ ESCAPE escape-string ] ]
```

Each column in a report can use zero or more of the above layout options. Each option can be used only once per column. The options are generally the same for headings and data, although some are not useful in headings. The options can be written in any order:

1. The WIDTH option overrides the default width for the data in the column. The default width is taken from the template in a USING statement, from the data dictionary for data elements, or from the string length for a literal without the REPEAT option. There is no default width for a variable. If the width is followed by IN (or INCH or INCHES), the width is measured in inches as determined by the horizontal pitch. The default is to measure width in characters.

2. The WIDTH UNLIMITED option formats the output without fixed column widths. This overrides the normal word wrap function, as well as the TRUNCATE, ALIGN, INDENT, and REPEAT formatting options. The data for the column is printed in the minimum space needed to contain the entire value, preceded by the optional SEPARATOR string. Numbers are formatted with decimal places defined in the usual manner, and with no leading spaces or zeros. This is useful with the PAGE WIDTH UNLIMITED option for producing output for export to other systems.

3. When LABEL is specified, the value printed is not the data, but the *elm_label* from the data dictionary for the specified element. This could be used in column headings. If LABEL is used with a literal or variable, the result is a blank.

4. The DIGITS option is used with numeric data to specify the number of digits to be printed to the left of the decimal point. The default number of digits for data elements is taken from the data dictionary. This specification is overridden if the WIDTH is given explicitly. Numeric values over 999 are printed with commas between groups of 3 digits. The commas are not counted as digits, but do count in the column width.

5. The DECIMAL option is used with numeric data to specify the number of digits to be printed to the right of the decimal point. The default number of decimal places is taken from the data dictionary if a data element is being printed, otherwise the default is zero. If the number of decimal digits is zero, the decimal point is not printed.

6. The SIGDIG option is used with numeric data to specify the number of significant digits in the value to be printed. The value is rounded off so that only the significant digits are shown, and zeros are added as necessary to fill out the remaining places required by the DIGITS and DECIMAL specifications. The number of significant digits specified must be greater than zero, and if SIGDIG is not specified, all digits are considered significant. The following examples show the relationship of the DECIMAL and SIGDIG specifications:

Original Value	DECIMAL	SIGDIG	Result
527.36	2	3	527.00
0.456	2	1	.50
1384.2	0	2	1400

7. The ALIGN option positions the data within the column. The default is based on the data dictionary definition for elements. For variables and literals the defaults are left alignment for character data and right for numeric.

8. The PAD option provides a character to fill out blank space in the column when the data is shorter than the column width. Padding occurs on the right if the column is aligned left, on the left if the column is aligned right, and on both sides if the column is aligned center. If the text in a column is word wrapped, padding is only applied on the last line of the text. The default pad character is a space.

9. The INDENT option positions the data a specified number of characters or inches from its alignment position. A positive indent applies to the first line of a word wrapped string, while a negative indent applies to lines after the first. In other words, for typical left aligned data, a positive indent produces first line indentation, while a negative indent produces "hanging" indentation. For right aligned data, it works the same way but relative to the right edge of the column.

10. The NEST option is provided for printing interpretations. These are traditionally printed in a "nested" or "outline" format, with results of each sub-rule indented below its parent rule. The amount of indentation increases with each level of sub-rule. So a nested format is really a variable indentation, with the amount of indent proportional

to the depth of nesting. The output of the INTERPRET command includes a column *rule_depth* for just this purpose. The NEST format option allows you specify an indentation of *n* spaces (or inches) per depth level. This can be combined with normal indentation, such as a negative indent amount for hanging indents. For example, to print an interp result *interphrc* with hanging indent of 1 space for word wrapping, plus nesting of 2 spaces per level, use:

```
interphrc INDENT -1 NEST 2 PER rule_depth
```

11. The NO COMMA option suppresses the placement of commas in numbers larger than three digits. This is used when printing a numeric value that should not have commas, such as a year or an id number. It is also used when exporting data in a comma delimited format, to avoid inappropriate commas.

12. The TRUNCATE option determines what happens when character type data is too long to fit in a column. The default is to split the data across multiple lines with word wrapping. If TRUNCATE is specified, the data is printed on a single line and truncated to the fit the column width. Numeric type data is never wrapped; if it is too long to fit the column, asterisks are printed.

13. The REPEAT option means that the column's value is repeated as often as necessary to fill the column width. This is typically used with a literal, such as "-". The REPEAT option in this case would fill the column with dashes. The column width must be specified explicitly when REPEAT is used.

14. The SEPARATOR string prints before the data for the column prints. It is placed at the column's specified starting position, and the actual data for the column starts after the separator. If this option is not included, no separator is printed. There is no way to specify a separator to the right of a field. To print a right border on a line, add a field of width zero at the end of the line, with the desired border character as its separator.

15. The REPLACE options allow the printing of some other value when a zero or null is found. This does not affect the operation of any calculations based on the value being replaced. This function can also be achieved using a variable with a conditional expression, but the REPLACE form might be more convenient. A value set to null by SUPPRESS DUPLICATES is not replaced with the substitution value, but always prints blank.

16. The SUPPRESS DUPLICATES option prevents repetitive printing of data. For each report input record, the value of a column specified with SUPPRESS is compared to its value in the previous record. If it matches, blanks print instead of the value. If the column is part of the sort key for the main report query, the duplicate suppression does not occur on control breaks. In this context, a control break occurs when the column or any column higher in the sort key changes value.

This control break behavior can be obtained for non-sort columns with the BY phrase. The identifier after BY is an element or variable to be tested if the value of the column itself does not change. If there is a change in the value of the BY variable (or higher sort columns if the BY variable is in the sort key), suppression does not occur.

17. The QUOTE or QUOTED option surrounds the column's data with quotation marks and escapes any embedded quotation marks. This is typically used when exporting data to another program. The quote-string is a single character that will be added to the beginning and end of the data. It defaults to the quotation mark ("). The escape-string is another single character whose default is the back-slash (\). If the data contains an occurrence of the quote-string or the escape-string, it will be preceded in the output by the escape-string. If the quote-string and the escape-string are the same, it means that embedded quotes will be doubled (which is the Informix convention). To specify a quotation mark, surround it by single quotes: '"'.

    For example, to use the single quote instead of the double quote, and to double the quote if it appears in the text, the format option would be written as:
    ```
    QUOTE "'" ESCAPE "'"
    ```
    If the original text contained the following:
    ```
    Elmer said, "That's all, folks."
    ```
    The output using the above format would be:
    ```
    'Elmer said, "That''s all, folks."'
    ```

Syntax:
SET *column_name* [FROM variable] [, *column_name* [FROM variable]] … .

Used In:
Calculation

Example:
```
SET aashind_l, aashind_r, aashind_h.
SET dbfifteenbar_r FROM db.
```

The SET statement is used in calculation scripts to store the results of a calculation back to the database. The value of the variable in the calculation script is placed in the specified column. If the column and variable have the same name, the FROM variable part may be omitted. One or more column assignments can be specified in a SET statement, and a script may contain more than one SET statement. In either case, the results are stored for each row that has been chosen to be calculated by the user. Rows that are locked or protected (not editable by the user) are never modified. If the specified column contains manually entered data, it is not changed unless the user chooses to override manually entered data (in the Calculation Manager dialog).

Values can be stored in two ways: singly or in groups. If the *column_name* is a column in the base table of the calculation, a single value will be stored in each calculated row. If the source variable has more than one value, only the first one is used. If the calculated variable has a null value, a null is stored in the column. Any time a value is stored, the source column associated with *column_name* is set to "Calculated".

A group of values can be stored by specifying a column in a table that is a direct child of the base table. This causes all existing child rows in the selected set to be updated with new data in the specified column. If necessary, rows will be added or deleted to match the number of values in the source variable. More than one column in the child table may be given new values, by using multiple SET statements or multiple columns in one SET. Case should be taken to see that all source variables have the same dimension, or data could be lost.

In case of ambiguity in column names, the form *table.column* may be used for *column_name*. The _r, _l, or _h suffix must be used if the column has modal values.

Syntax:
TEMPLATE template-name [column-layout] line-specification **.**

template-name ⇒ *name*

Used In:
Report

Example:
```
TEMPLATE basic SEPARATOR "|"
AT LEFT FIELD WIDTH 8, FIELD WIDTH 50.
```

A template describes the format of a report line without the data. Templates are not required, but are useful to avoid repetitive specification of layout options. Putting the statement "USING template-name" into a line specification copies all the column layout information from the template into the line specification.

In a template, a set of column layout options can be given right after the template name, and these will be the default for all columns in the template. There must be one and only one line specification in the form "AT position column-spec ...". This can contain additional column layout options, which take precedence over the template defaults. Finally, when a template is invoked with a USING statement, other layout options can be given, which take precedence over the template. Column and line specifications are described under the SECTION statement.

In the line specification used in a template, it is possible to use a literal, variable or element name as a value to be printed in some column. This would print the specified value whenever the template is used. However, the keyword FIELD can also be used in place of a value, which means that the value to be printed is not defined until it is specified in a USING statement. A line specification in a template definition may not contain USING.

Syntax:
WHEN expression DISPLAY message [parameter [, parameter] …] **.**

Used In:
Validation

Example:
```
WHEN sum_pct > 100 DISPLAY "Percents sum to more than 100".
WHEN error DISPLAY "Error in horizon %s" hzname.
```

The WHEN statement is used in validation scripts to produce a message when an error condition is detected. The expression after WHEN is evaluated for each row to be validated, and if a True (non-zero) value is found the message is added to the validation message list. If the message contains substitution markers as used in *sprintf* (such as %s or %g) values are taken from the list of parameters and placed into the message. The validation process also records information about which row generated a message, and this is included when the message list is displayed.

In some cases it is useful to have multiple values for the WHEN expression, or for the message or its parameters. This causes multiple messages to be generated for each row validated. If the validation script extracts data from a child of the base table, individual messages for each child row can be produced by using parameters that have values collected from the child rows.

NASIS CVIR Script Writing References

Database Structure Guide

The NASIS 5.0 Database Structure Guide is a comprehensive reference that describes all aspects of the NASIS database design. The guide provides information you need to know about the NASIS template model, naming conventions and data types. It can be obtained from the NASIS web site: http://nasis.nrcs.usda.gov/documents/metadata/5_0.

Table Structure Report

The Table Structure Report is included in the *NASIS 5.0 Database Structure Guide*. The table structure report provides information you need to know about table and column physical names, modality, data types, and other characteristics necessary for report writing.

Database Structure Diagrams

The Database Structure Diagrams are included in the *NASIS 5.0 Database Structure Guide* and the NASIS Online Help. Because of the size of the database the diagrams each show just one object hierarchy. They show the table relationships required for completing joins between tables.

Related Reading

Informix Software, Inc. 1999. Advanced select statements. *In* The Informix Guide to SQL:Tutorial. Informix Software, Inc. Menlo Park, California. *refer to Chapter 3 for discussion about* OUTER *joins*

Informix Software, Inc. 1999. Guide to SQL:Tutorial. Available in .pdf format at http://www.informix.com/answers. Informix Software, Inc. Menlo Park, California.

Valley, John J. 1991. UNIX Programmer's Reference. Que Corporation. Carmel, Indiana. *refer to the C Library function* printf *on page 570 for* sprintf *syntax*.

Changes in NASIS 5.1 CVIR Functions

This book, the *NASIS 5.1 CVIR Script Writing Technical Reference*, describes the use of all CVIR functions in detail. The following list includes changes made in version 5.0 and later releases. For a more complete description of differences between the NASIS 5.1 functions and previous functions, including the reasons for changes, see the NASIS technical website at http://nasis.nrcs.usda.gov/products.

New Features

ROUND function	New function for use in DEFINE statements to round off a number to a specified number of decimal places.
COUNT function	Keeps a running count of non-null values in a variable.
ARRAYCOUNT function	Counts the non-null values in an array of data.
ARRAYMEDIAN function	Finds the median value in an array.
ARRAYMODE function	Finds the modal value in an array.
ARRAYSTDEV function	Finds the standard deviation of the values in an array.

Revised Features

INTERPRET BY	Allows the base table for interpretations to be different from the base table for a report.
Subreport improvements	Some limitations in the previous implementation of subreports have been removed. Subreports can now be called recursively, and the DERIVE statement can be used in subreports.
PAGE PAD	This was originally a section type, and did not work consistently. It is now an independent page format specification and is defined at the beginning of report execution. Now the page padding will print the same way regardless of where it is used.
Vertical Alignment	Line specifications have been enhanced to permit control over the placement of AT groups relative to their adjacent AT group. See the section on Line Specifications for details.

Index